



MCUXpresso SDK Documentation

Release 25.12.00



NXP
Dec 18, 2025



Table of contents

1	KW47-LOC	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	18
1.3.1	Getting Started with MCUXpresso SDK Repository	18
1.4	Release Notes	43
1.4.1	MCUXpresso SDK Release Notes	43
1.5	ChangeLog	53
1.5.1	MCUXpresso SDK Changelog	53
1.6	Driver API Reference Manual	147
1.7	Middleware Documentation	147
1.7.1	Wireless Bluetooth LE host stack and applications	147
1.7.2	Wireless Connectivity Framework	147
1.7.3	Multicore	148
1.7.4	FreeMASTER	148
1.7.5	FreeRTOS	148
2	KW47B42ZB7	149
2.1	CACHE: LPCAC CACHE Memory Controller	149
2.2	CCM32K: 32kHz Clock Control Module	150
2.3	CE Basic Functions	160
2.4	CE Command Functions	160
2.5	CE CMSIS Functions	162
2.6	CE Initialization Functions	162
2.7	CE Matrix Functions	163
2.8	CE Transform Functions	167
2.9	Clock Driver	169
2.10	CMC: Core Mode Controller Driver	189
2.11	CRC: Cyclic Redundancy Check Driver	201
2.12	EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	204
2.13	ELEMU: Edgelock Messaging unit driver	223
2.14	EWM: External Watchdog Monitor Driver	227
2.15	FGPIO Driver	230
2.16	C40ESP3 Flash Driver	230
2.17	FlexCAN: Flex Controller Area Network Driver	233
2.18	FlexCAN Driver	233
2.19	FlexCAN eDMA Driver	282
2.20	FlexIO: FlexIO Driver	284
2.21	FlexIO Driver	284
2.22	FlexIO eDMA SPI Driver	302
2.23	FlexIO eDMA UART Driver	305
2.24	FlexIO I2C Master Driver	308
2.25	FlexIO I2S Driver	316
2.26	FlexIO SPI Driver	327
2.27	FlexIO UART Driver	340

2.28	GPIO: General-Purpose Input/Output Driver	350
2.29	GPIO Driver	353
2.30	I3C: I3C Driver	359
2.31	I3C Common Driver	361
2.32	I3C Master Driver	363
2.33	I3C Slave Driver	389
2.34	IMU: Inter CPU Messaging Unit	402
2.35	Common Driver	409
2.36	LIN: Local Interconnect Network Driver	421
2.37	LIN Driver	421
2.38	LIN LPUART Driver	437
2.39	LPADC: 12-bit SAR Analog-to-Digital Converter Driver	444
2.40	LPCMP: Low Power Analog Comparator Driver	465
2.41	LPI2C: Low Power Inter-Integrated Circuit Driver	471
2.42	LPI2C Master Driver	473
2.43	LPI2C Master DMA Driver	487
2.44	LPI2C Slave Driver	489
2.45	LPIT: Low-Power Interrupt Timer	500
2.46	LPSPi: Low Power Serial Peripheral Interface	506
2.47	LPSPi Peripheral driver	506
2.48	LPSPi eDMA Driver	528
2.49	LPTMR: Low-Power Timer	535
2.50	LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver	540
2.51	LPUART Driver	540
2.52	LPUART eDMA Driver	559
2.53	LTC: LP Trusted Cryptography	562
2.54	LTC AES driver	563
2.55	LTC DES driver	568
2.56	LTC HASH driver	577
2.57	LTC PKHA driver	578
2.58	LTC Blocking APIs	587
2.59	MCM: Miscellaneous Control Module	587
2.60	MSCM: Miscellaneous System Control	592
2.61	MU: Messaging Unit	593
2.62	PORT: Port Control and Interrupts	605
2.63	RTC: Real Time Clock	613
2.64	SEMA42: Hardware Semaphores Driver	620
2.65	SFA: Signal Frequency Analyser	623
2.66	SMSCM: Secure Miscellaneous System Control Module	632
2.67	SPC: System Power Control driver	637
2.68	SYSPM: System Performance Monitor	672
2.69	TPM: Timer PWM Module	675
2.70	TRDC: Trusted Resource Domain Controller	691
2.71	TRGMUX: Trigger Mux Driver	712
2.72	TSTMR: Timestamp Timer Driver	713
2.73	VBAT: Smart Power Switch	714
2.74	VREF: Voltage Reference Driver	721
2.75	WDOG32: 32-bit Watchdog Timer	724
2.76	WUU: Wakeup Unit driver	730
3	Middleware	737
3.1	Motor Control	737
3.1.1	FreeMASTER	737
3.2	MultiCore	774
3.2.1	Multicore SDK	774
3.3	Wireless	872
3.3.1	NXP Wireless Framework and Stacks	872

4	RTOS	947
4.1	FreeRTOS	947
4.1.1	FreeRTOS kernel	947
4.1.2	FreeRTOS drivers	947
4.1.3	backoffalgorithm	947
4.1.4	corehttp	947
4.1.5	corejson	947
4.1.6	coremqtt	948
4.1.7	corepkcs11	948
4.1.8	freertos-plus-tcp	948

This documentation contains information specific to the kw47loc board.

Chapter 1

KW47-LOC

1.1 Overview

The KW47 LOC is an evaluation kit for KW47 Automotive MCU with 2.4 GHz Bluetooth Low Energy and generic FSK wireless connectivity and dual CAN/LIN connectivity. This localization platform is dedicated to Bluetooth Channel Sounding Ranging solution development with a dedicated on-chip Localization Compute Engine to reduce ranging latency. The KW47's highly sensitive, optimized 2.4 GHz radio features is featured with dual antenna diversity via RF switch which can be bypassed to test via SMA RF connection. The board includes an advanced MCU-Link debug probe, Power and low power option, dual CAN transceiver, buttons, switches, LEDs and a MikroE Click connector.



MCU device and part on board is shown below:

- Device: KW47B42ZB7
- PartNumber: KW47B42ZB7AFTA

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

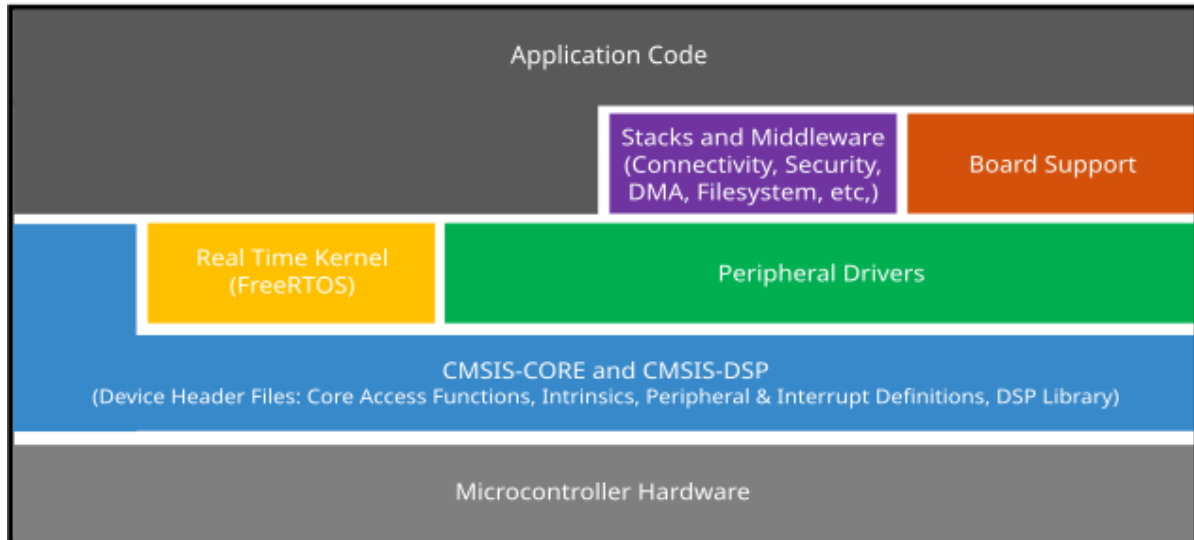
Overview

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

also contains optional RTOS integrations such as FreeRTOS and Azure RTOS, device stack, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for KW47-LOC*.

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



MCUXpresso SDK board support package folders

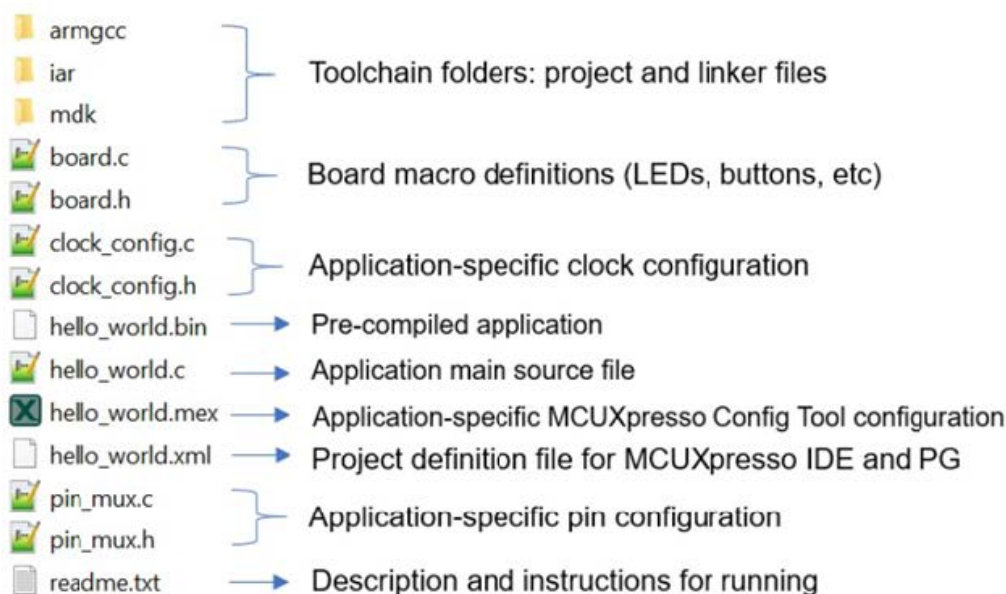
MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top-level boards folder and each supported board has its own folder (an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

- `demo_apps`: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may use stacks and middleware.
- `driver_examples`: Simple applications that show how to use the peripheral drivers of the MCUXpresso SDK 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 RTOS drivers for the MCUXpresso SDK.
- `wireless_examples`: Applications that use the Wireless stacks.

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

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

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

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

Locating example application source files When opening an example application in any of the supported IDEs, various 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>`: CMSIS header file of the device, MCUXpresso SDK feature file, and a few other files
- `devices/<device_name>/drivers`: All the peripheral drivers for your specific MCU
- `devices/<device_name>/<tool_name>`: Toolchain-specific startup code, including vector table definitions
- `devices/<device_name>/utilities`: Items such as the debug console that are used by many of the example applications
- `devices/<device_name>/project_template`: Project template used by MCUXpresso IDE to create projects

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

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

Running a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the **kw47loc** hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

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

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

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

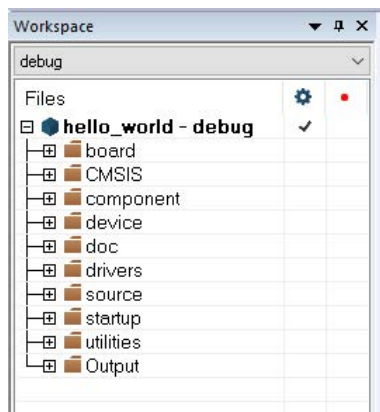
Using the kw47loc hardware platform as an example, the `hello_world` workspace is located in:

```
<install_dir>/boards/kw47loc/demo_apps/hello_world/iar/hello_world.eww
```

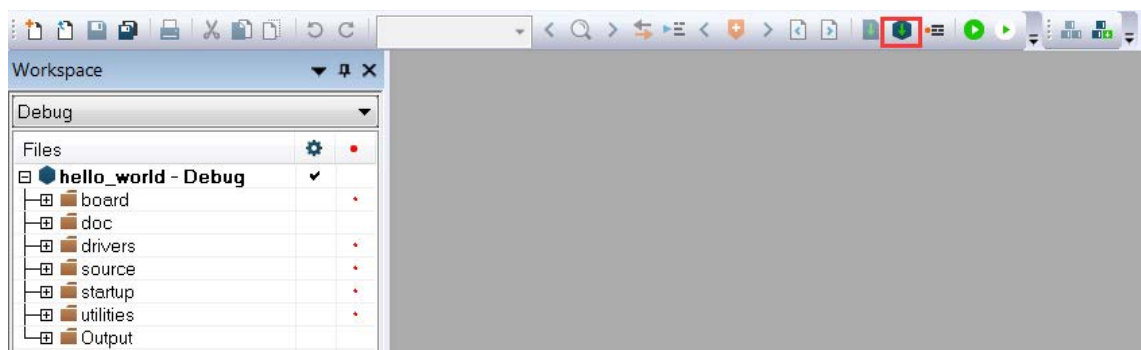
Other example applications may have additional folders in their path.

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

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



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

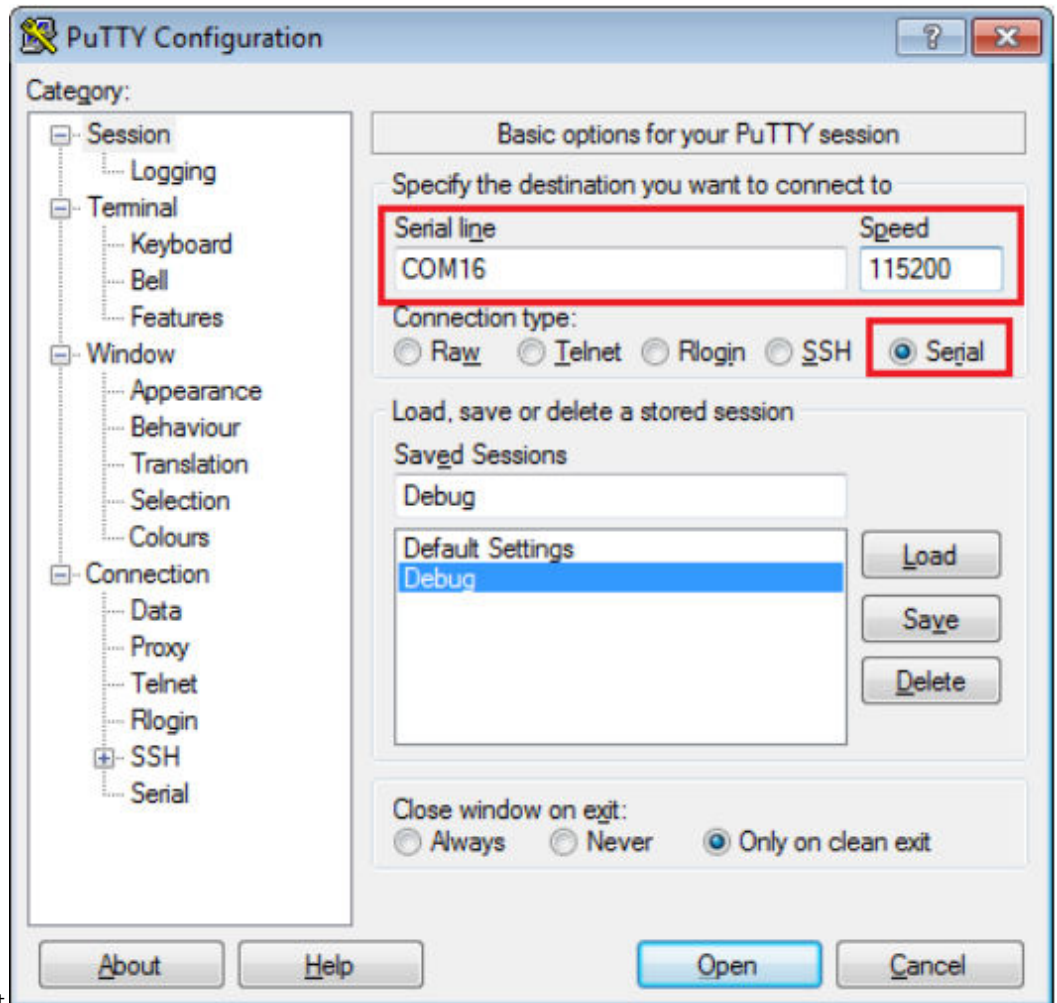


4. The build completes without errors.

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

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

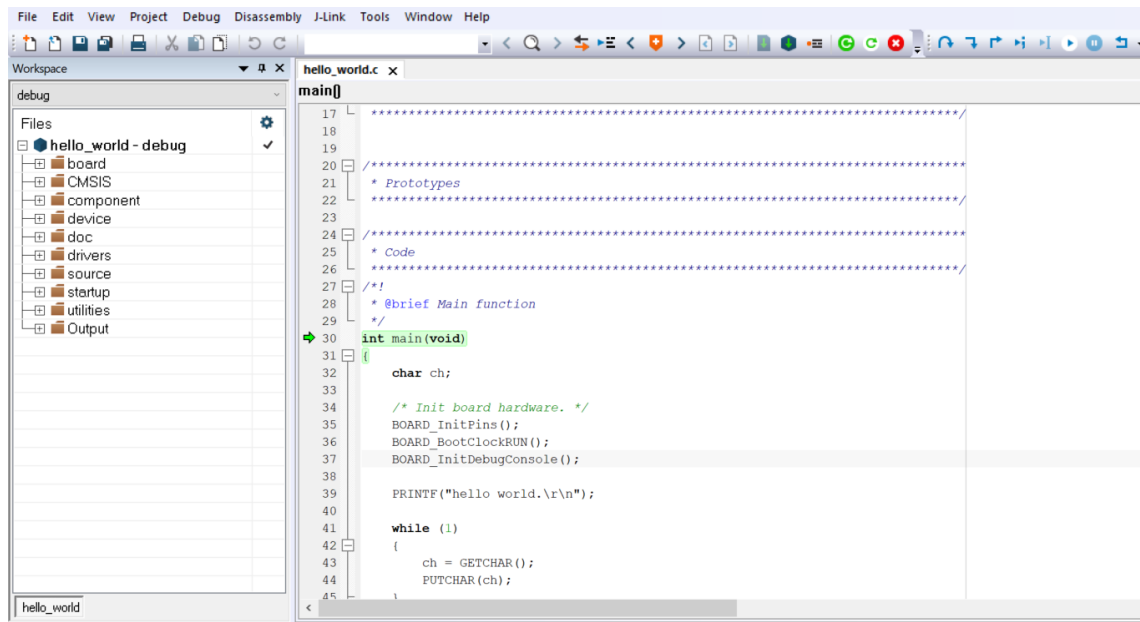
1. Connect the development platform to your PC via USB cable between the USB connector (J14) and the PC USB connector.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port. To determine the COM port number, see [How to determine COM Port](#). Configure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



4. 1 stop bit
3. In IAR, click the **Download and Debug** button to download the application to the target.



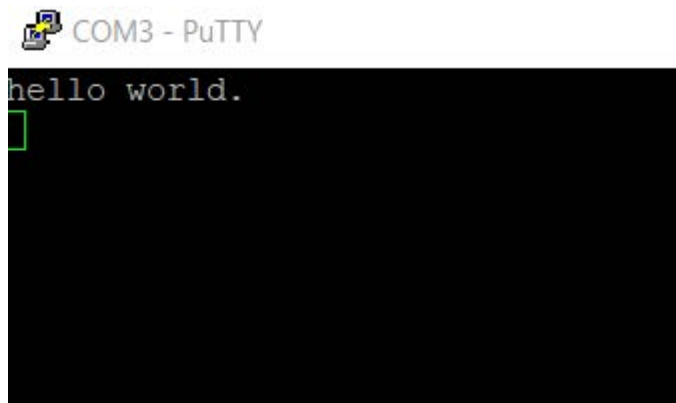
4. The application is then downloaded to the target and automatically runs to the `main()` function.



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



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



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

Run a demo using MCUXpresso IDE

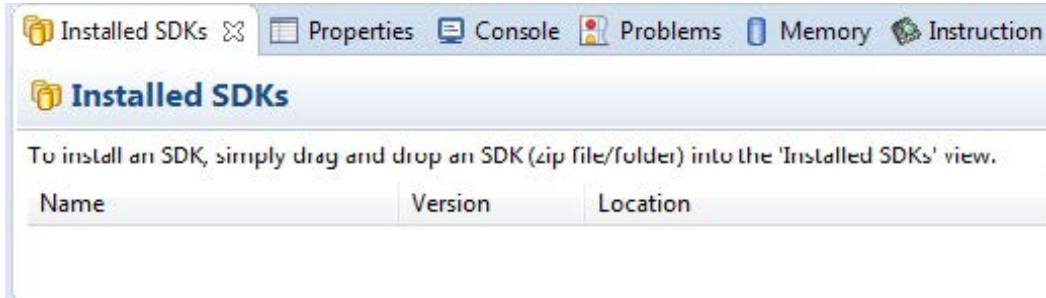
This section describes the steps required to configure MCUXpresso IDE to build, run, and debug example applications. The `hello_world` demo application targeted for the KW47-LOC hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

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

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

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

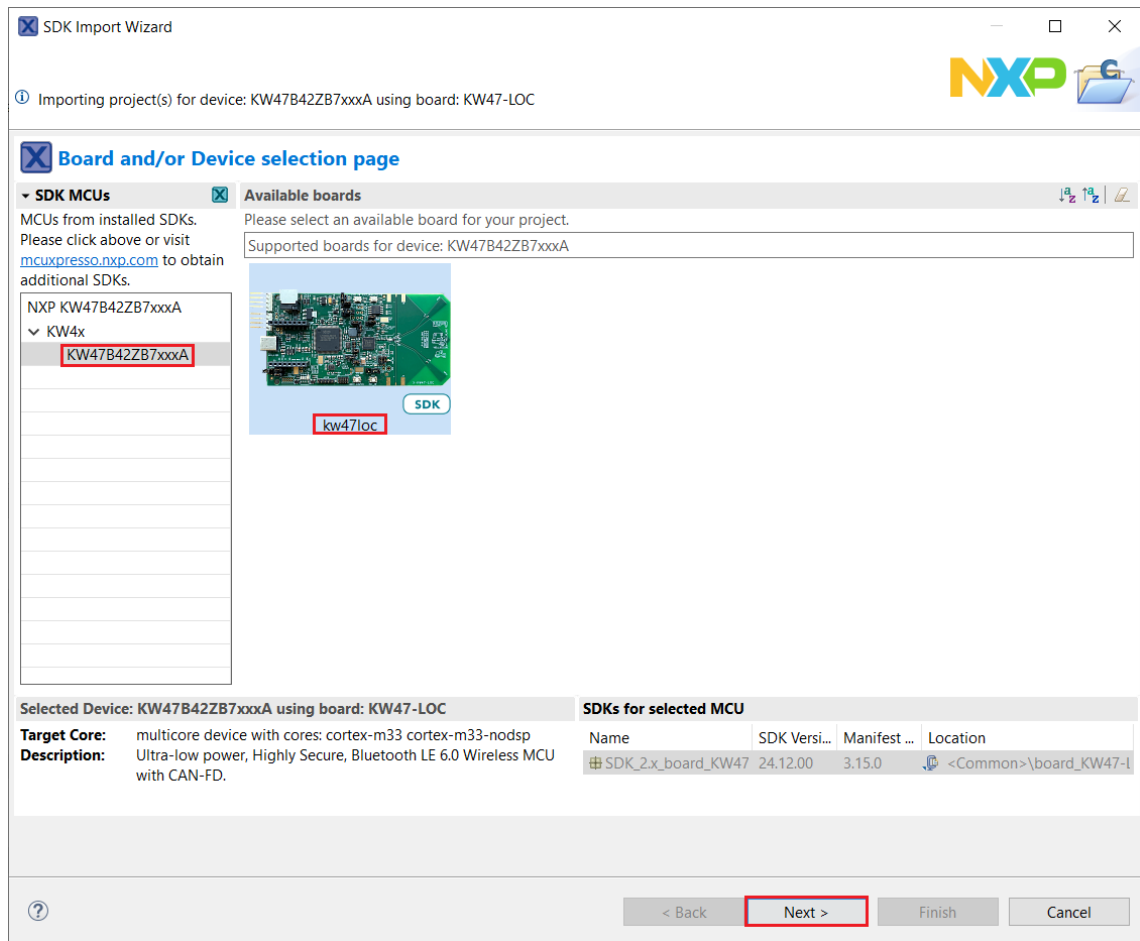
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



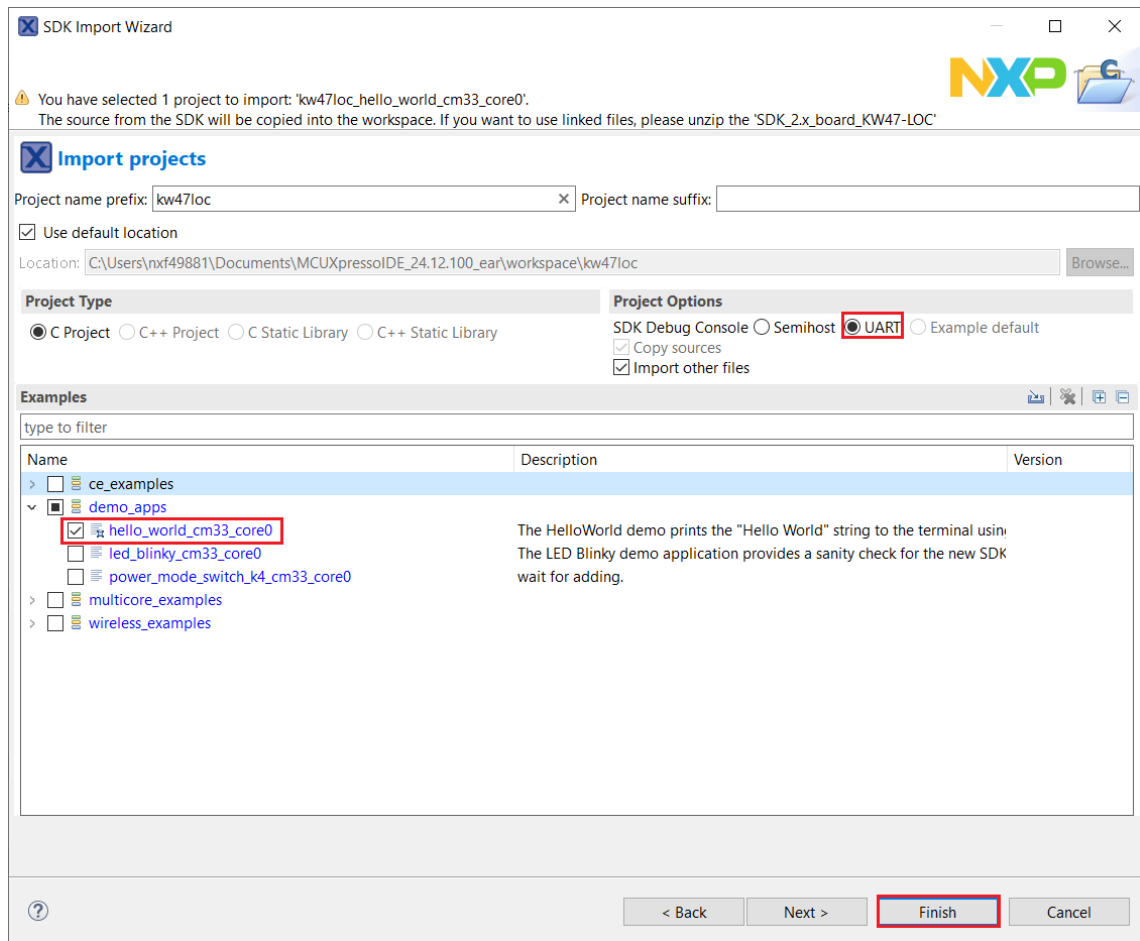
2. On the **Quickstart Panel**, click **Import SDK example(s)...**



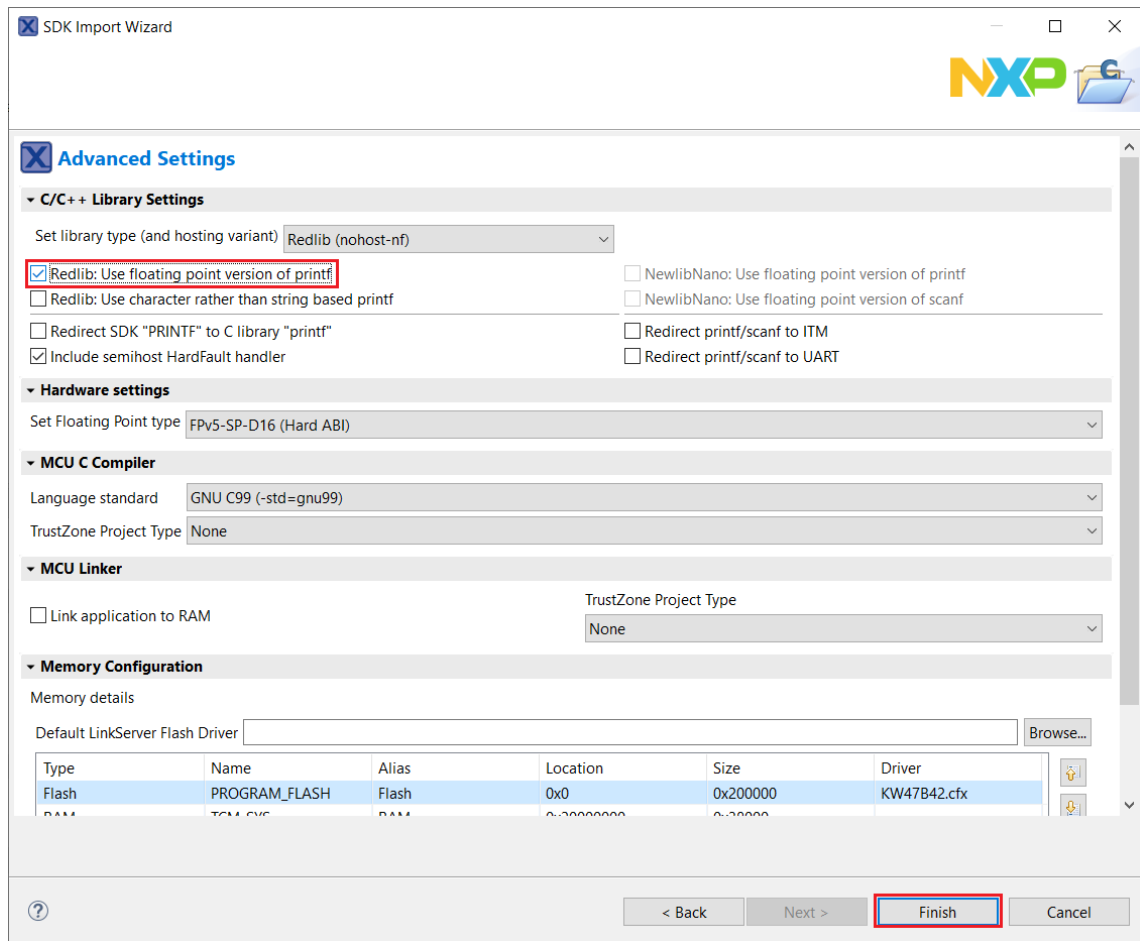
3. In the window that appears, expand the **KW4x** folder and select **KW47**. Then, select **kw47loc** and click **Next**.



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



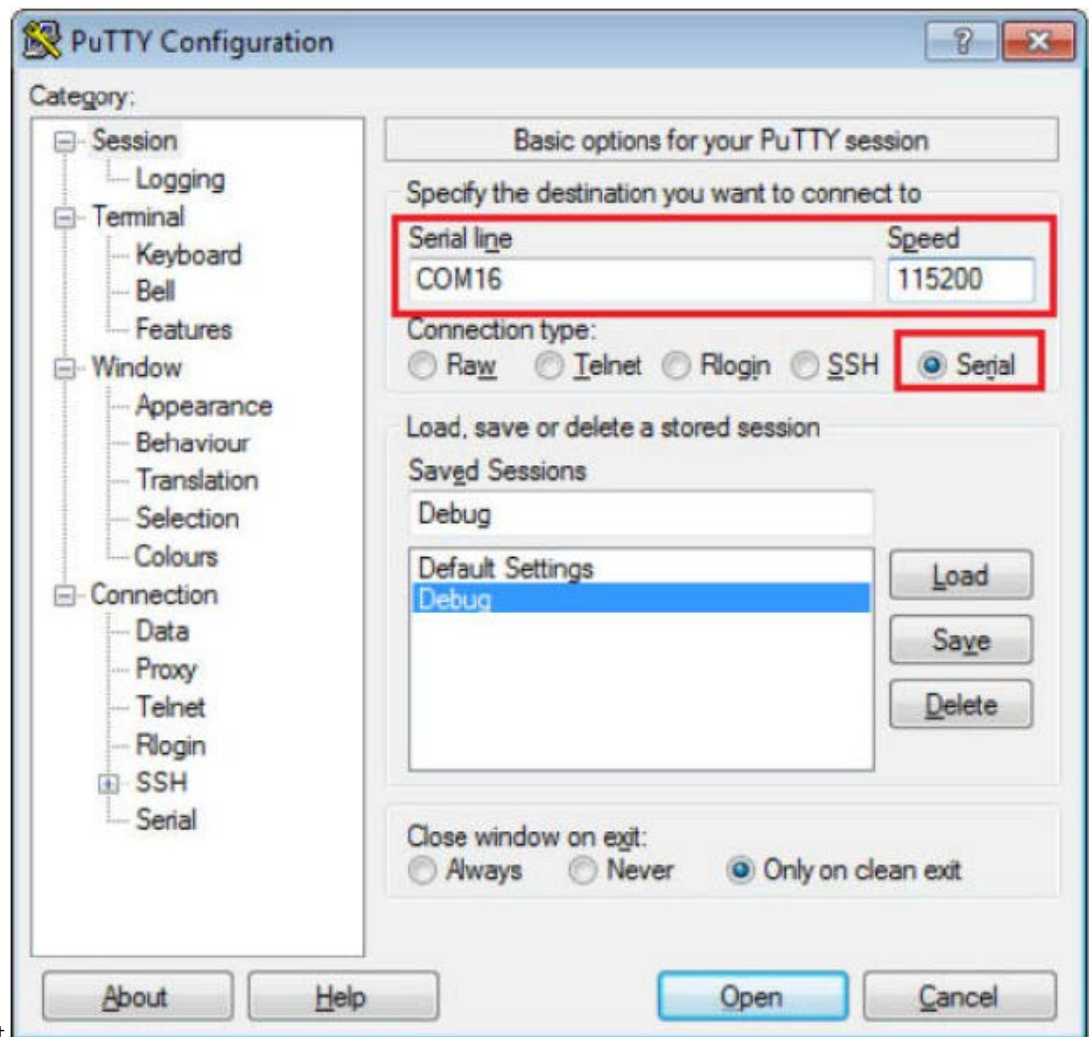
5. Ensure **Redlib: Use floating point version of printf** is selected if the example prints floating point numbers on the terminal for demo applications such as `adc_basic`, `adc_burst`, `adc_dma`, and `adc_interrupt`. Otherwise, it is not necessary to select this option. Then, click **Finish**.



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

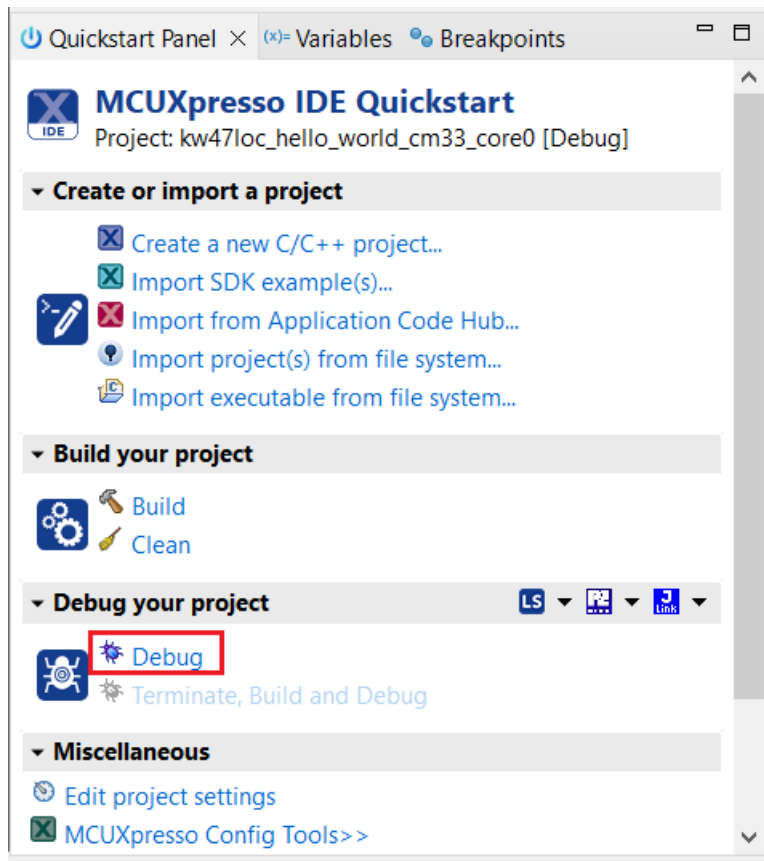
Running an example application To download and run the application, perform the following steps:

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

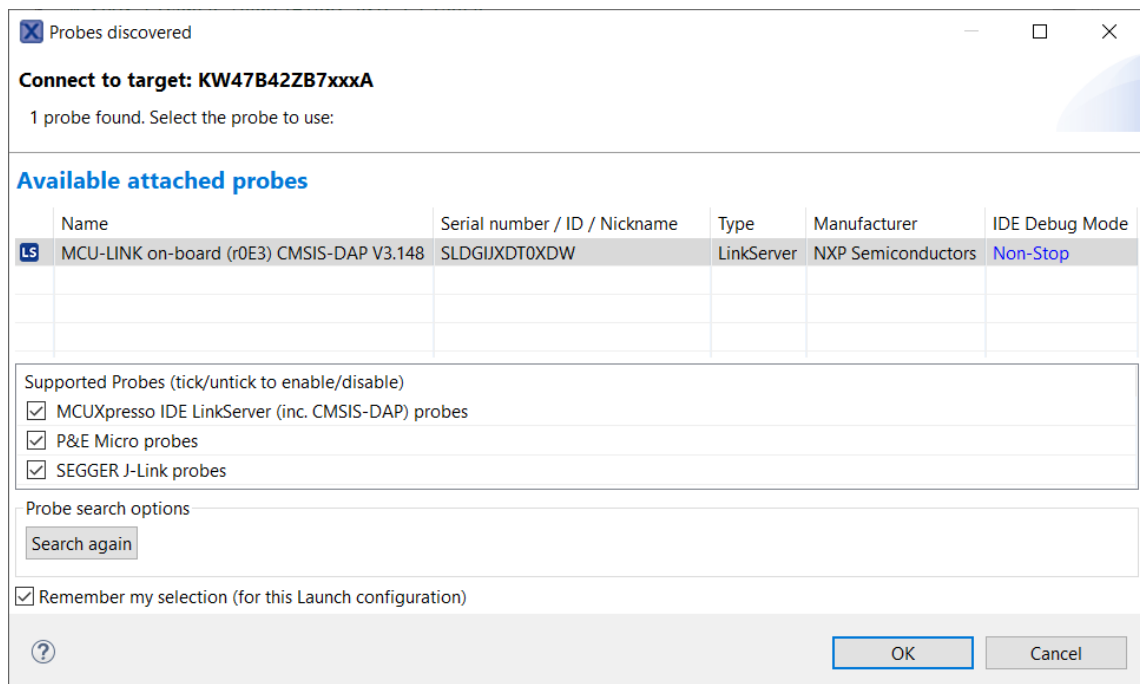


4. 1 stop bit

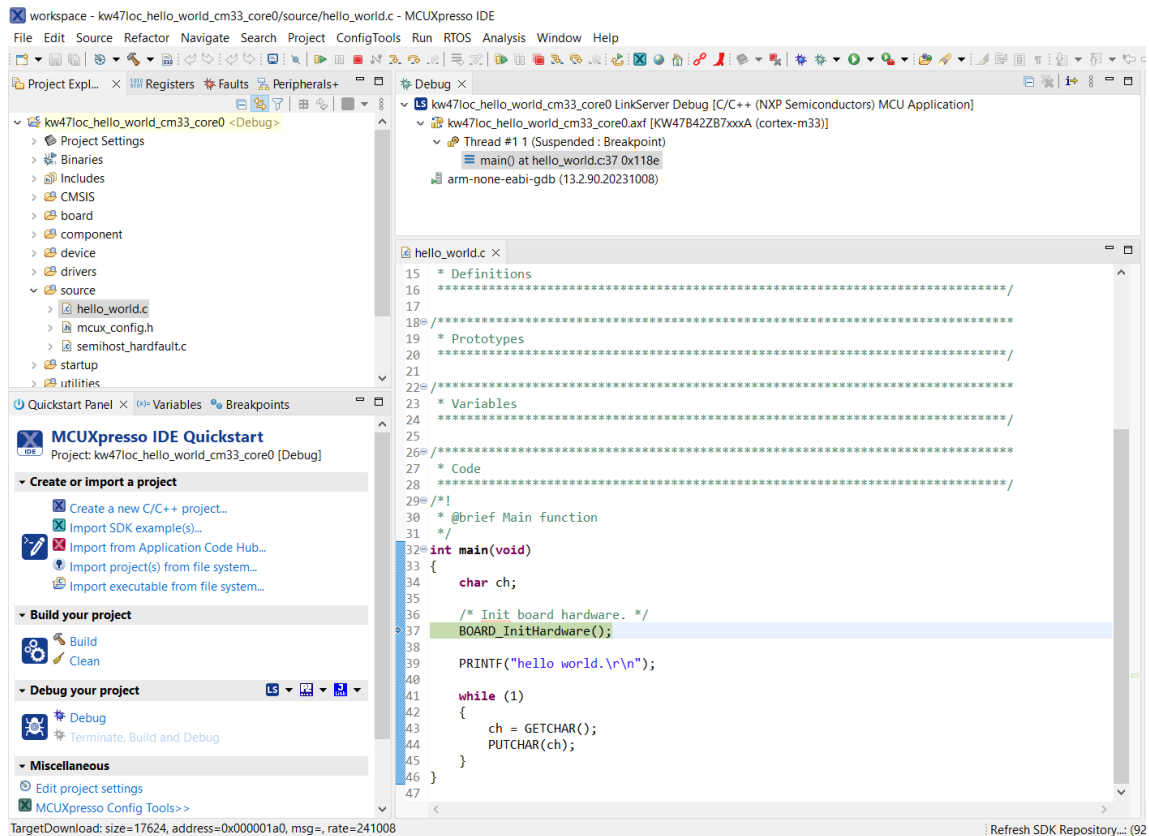
3. On the **Quickstart Panel**, click on **Debug**.



- 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.)



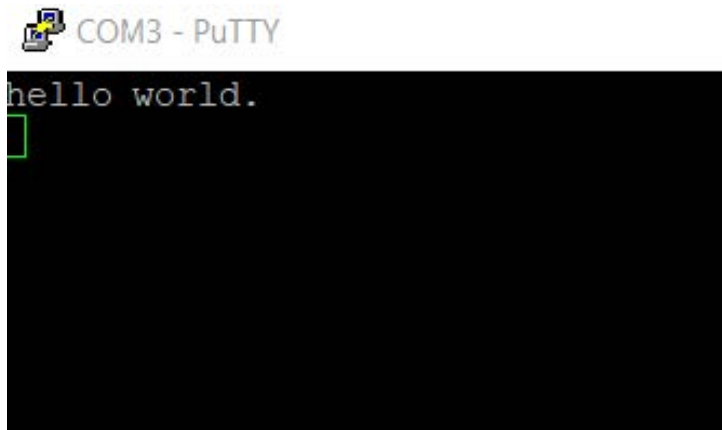
- The application is downloaded to the target and automatically runs to `main()`.



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



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

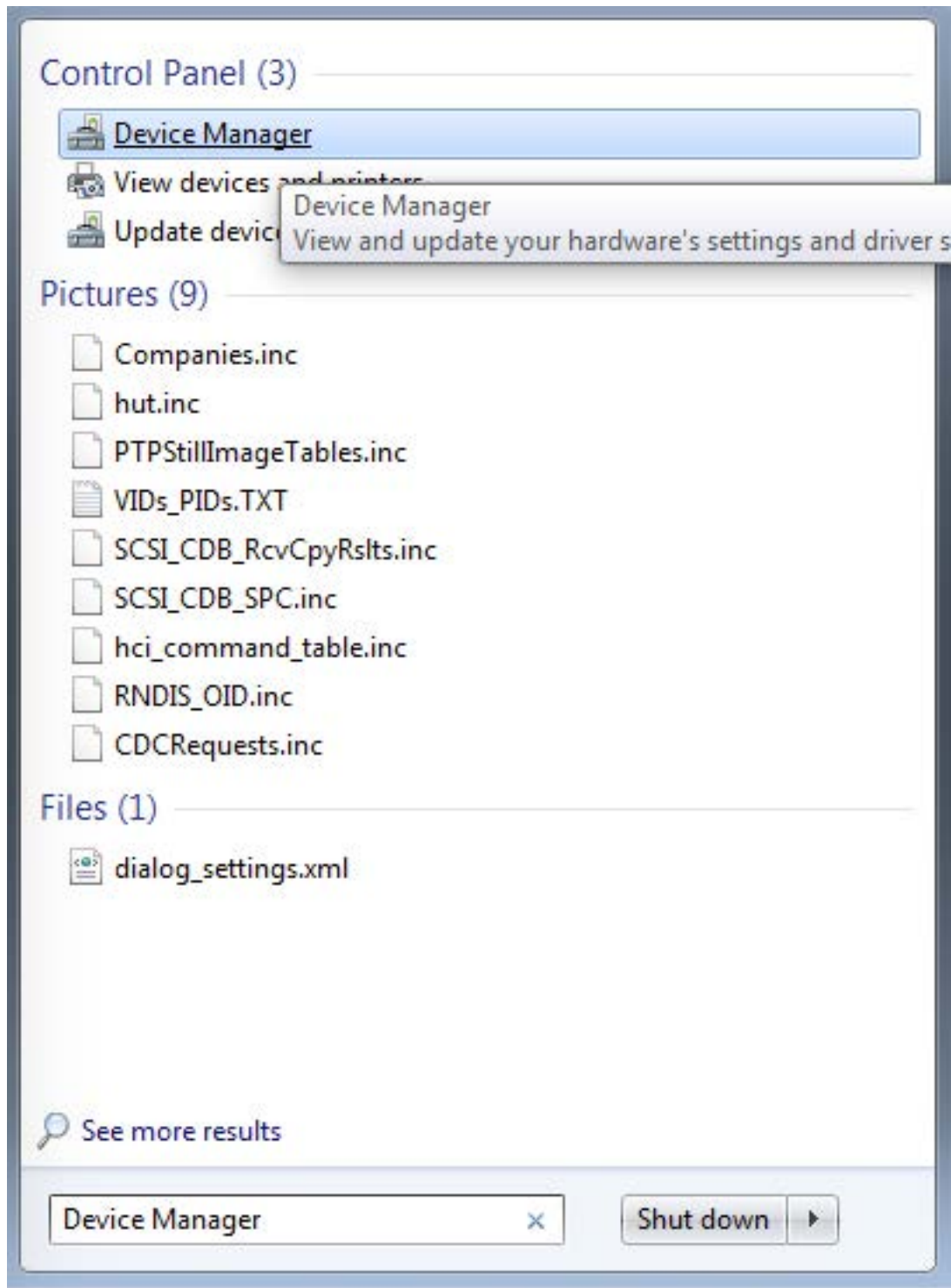


Parent topic: [Run a demo using MCUXpresso IDE](#)

How to determine COM Port

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

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



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

How to set the board to the Bootloader ISP mode

On the KW47-LOC board:

1. Press and hold the BOOT CONFIG switch, **SW4**.

2. Connect the KW47-LOC board via the micro-USB connector to the PC.
3. Release the switch **SW4**.
4. Check the associated USB port number on the PC (such as **COM10**).
5. While the KW47 is in bootloader ISP mode, navigate to the folder where *blhost.exe* is located.
6. Type the command `.\blhost.exe -p COM10 -- get-property 1` to make sure that the KW47 is in Bootloader ISP mode, you should see:
 - Ping responded in 1 attempt(s)
 - Inject command 'get-property'
 - Response status = 0 (0x0) Success.
 - Response word 1 = 1258488064 (0x4b030100)
 - Current Version = K3.1.0

On a custom board:

1. Short the `BOOT_CFG` pin to VDD while reset.

Updating debugger firmware

The KW47-LOC board comes with a CMSIS-DAP-compatible debug interface (known as MCU-Link). This firmware in this debug interface may be updated using the host computer scripts. This firmware is typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to re-program the debug probe firmware.

NXP provides the MCU-Link utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto MCU-Link. The utility can be downloaded from [MCU-Link Debug Probe](#).

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in MCU-Link user guide, [MCU-Link Debug Probe](#).

1. Install the MCU-Link utility.
2. Unplug the USB cable of the board.
3. Install the jumper on JP12.
4. Connect the probe to the host via USB (use Link USB connector).
5. Open a command shell and call the appropriate script located in the MCU-Link installation directory, *<MCU-Link install dir>*.
 1. To program CMSIS-DAP debug firmware: *<MCU-Link install dir>/scripts/program_CMSIS*.
 2. To program J-Link debug firmware: *<MCU-Link install dir>/scripts/program_JLINK*.
6. Remove the jumper on JP12.
7. Re-power the board by removing the USB cable and plugging it in again.

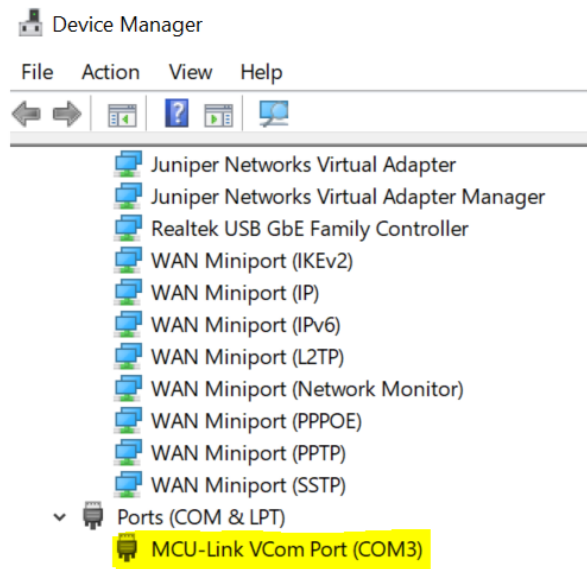
Updating NBU for Wireless Examples

The Narrow Band Unit (NBU) firmware is included in the SDK folder as a signed and encrypted FW.

middleware\wireless\ble_controller\bin\kw47_nbu_ble_hosted.bin

To program the NBU software for the KW47, perform the following steps:

1. While holding pressed the **SW4** on the KW47-LOC board, attach the USB connector J3 to your computer. Then, release the **SW4** after you plugged the USB cable on your computer.
2. Verify which COM port is assigned to your KW47-LOC board. To check the assigned COM port, in the Windows **Device Manager** program, search for Ports (**COM & LPT**) and save the COM port number. In this example, the assigned COM port is **COM3**.



3. Execute the following commands in order, replacing **COMX** with your actual COM port number (e.g., **COM3**):

Make sure the device has entered ISP mode properly. If the device did not enter in ISP mode, it cannot be programmed.

```
blhost.exe -p COMX get-property 1
```

Erase NBU flash

```
blhost.exe -p COMX flash-erase-all 2
```

Program NBU binary into NBU flash

```
blhost.exe -p COMX write-memory 0x48800000 kw47_nbu_ble_hosted.bin
```

The following examples show when the device did not enter in ISP (left) and when the device was programmed properly (right).

```
Error: Initial ping failure: No response received for ping command.
```

```
Ping responded in 1 attempt(s)
Inject command 'get-property'
Response status = 0 (0x0) Success.
Response word 1 = 1258488064 (0x4b030100)
Current Version = K3.1.0
```

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 Compiler	ARM ecosystem Advanced optimization	ARM Developer ARM Developer	MDK_DIR 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
arch/	Architecture-specific files	ARM CMSIS, build configurations
cmake/	Build system modules	CMake configuration and build rules
compo	Software components	Reusable software libraries and utilities
devices	Device support packages	MCU-specific headers, startup code, linker scripts
drivers	Peripheral drivers	Hardware abstraction layer for MCU peripherals
examp	Sample applications	Demonstration code and reference implementations
middle	Optional software stacks	Networking, graphics, security, and other libraries
rtos/	Operating system support	FreeRTOS integration
scripts	Build and utility scripts	West extensions and development tools
svd	Svd files for devices, this is optional because of large size. Customers run <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

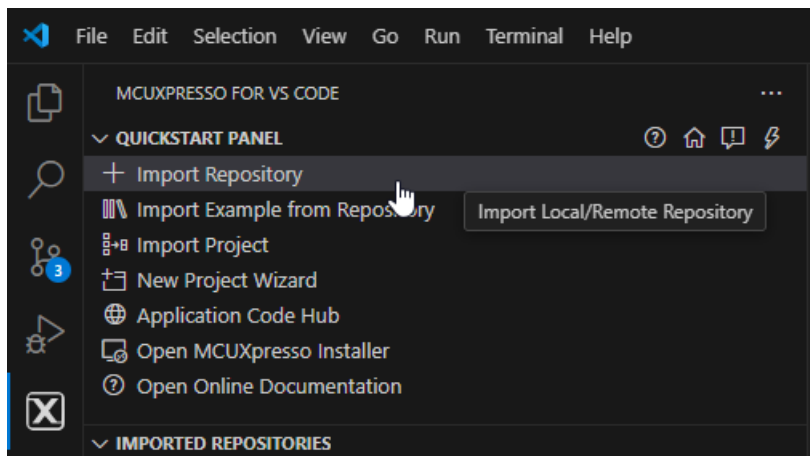
Extension Installation

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.

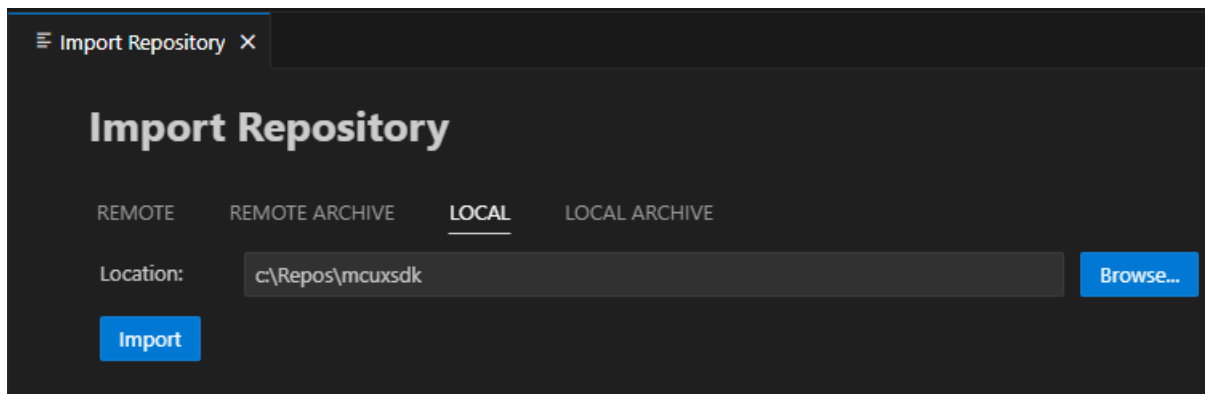
SDK Import and Setup

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.

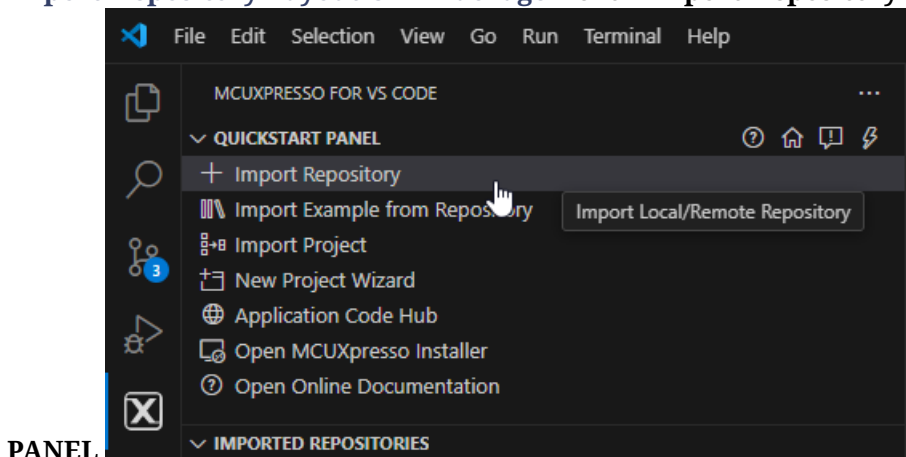
Import GitHub Repository SDK Click **Import Repository** from the **QUICKSTART PANEL**



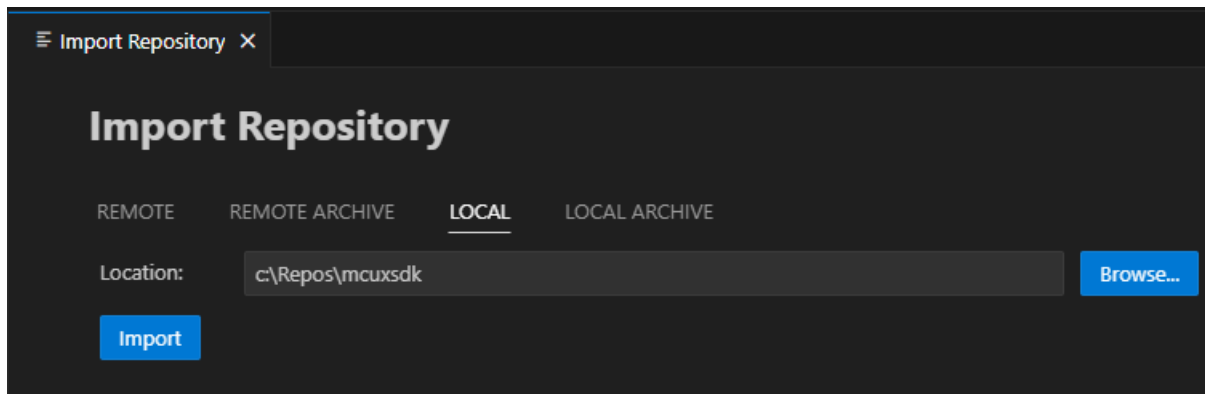
Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details. Select **Local** if you've already obtained the SDK according to [setting up the repo](#). Select your location and click **Import**.



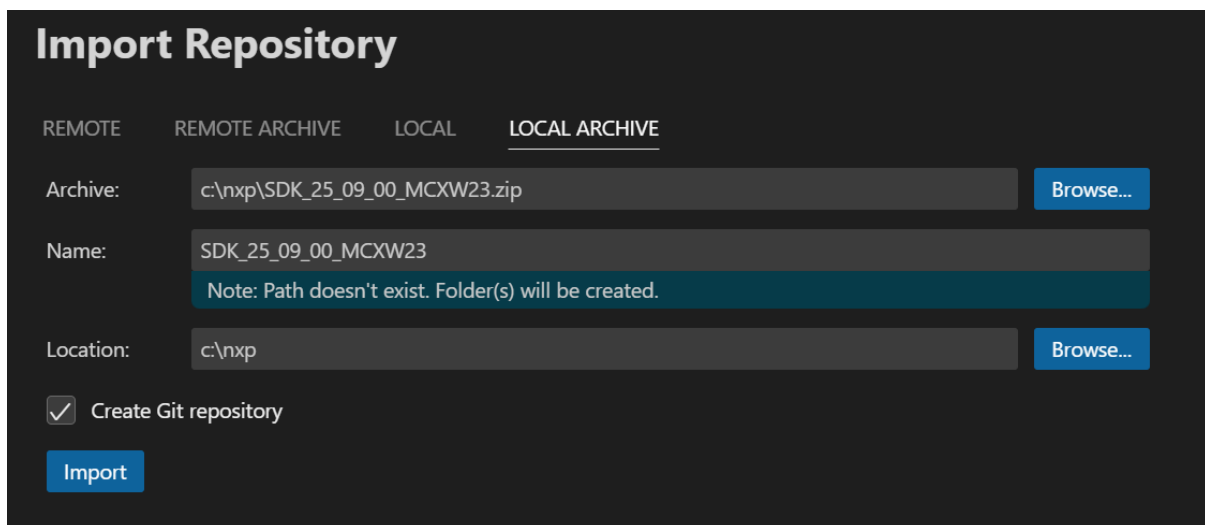
Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART**



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



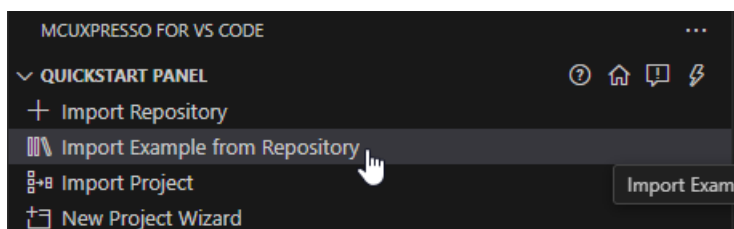
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



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:
frdmmcxc444_hello_world

Location:
c:\nxp_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

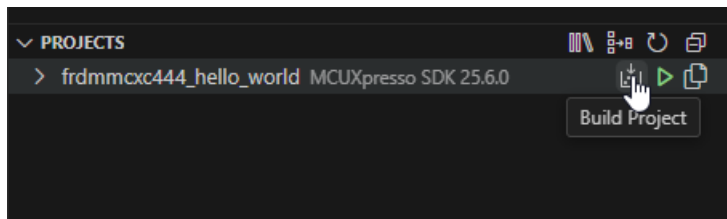
Trust Confirmation VS Code will prompt you to confirm if the imported files are trusted. Click **Yes** to proceed.

Building Projects

Build Process

1. Navigate to **PROJECTS** view

2. Find your project
3. Click the **Build Project** icon

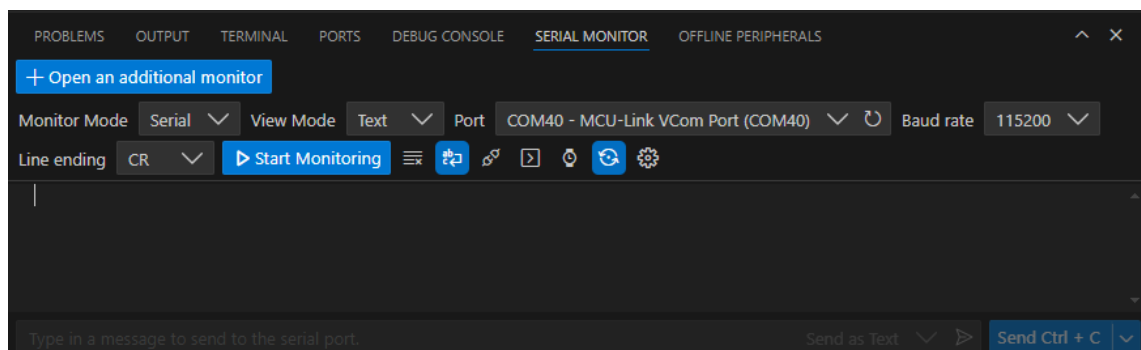


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

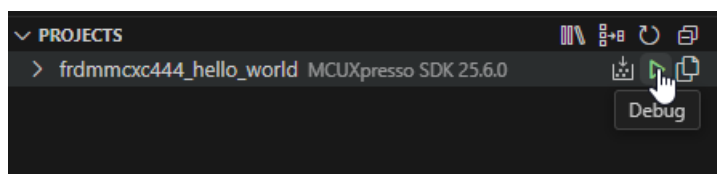
1. Open **Serial Monitor** from VS Code's integrated terminal



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

Debug Session

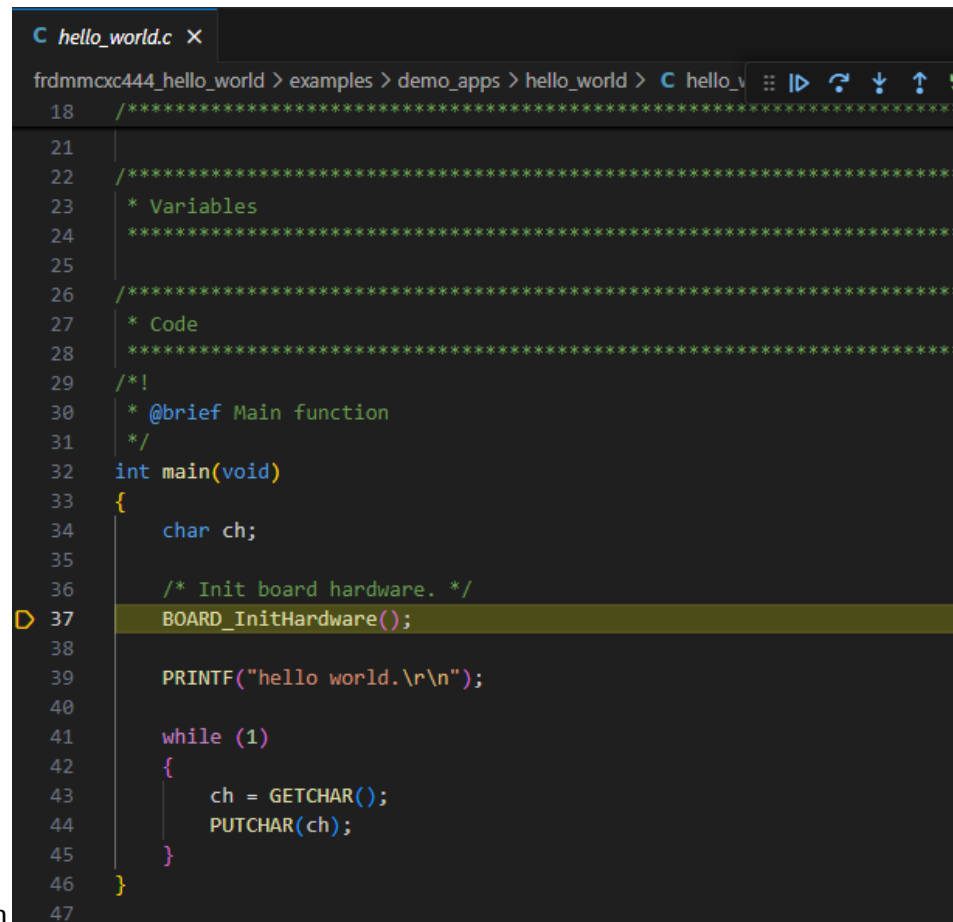
1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



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



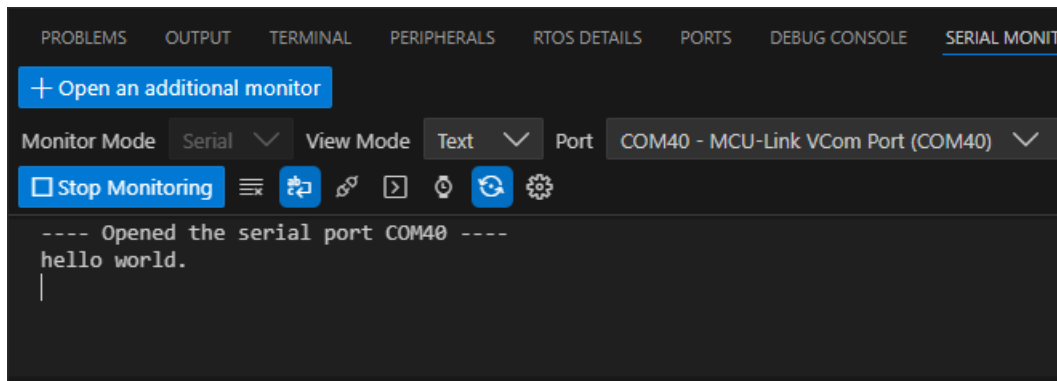
```

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

This is an early adopter release provided as preview for development with pre-production devices.

1.4.1 MCUXpresso SDK Release Notes

Overview

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

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

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

MCUXpresso SDK

As part of the MCUXpresso software and tools, MCUXpresso SDK is the evolution of Kinetis SDK, includes support for LPC, DSC, PN76, and i.MX System-on-Chip (SoC). The same drivers, APIs, and middleware are still available with support for Kinetis, LPC, DSC, and i.MX silicon. The MCUXpresso SDK adds support for the MCUXpresso IDE, an Eclipse-based toolchain that works with all MCUXpresso SDKs. Easily import your SDK into the new toolchain to access to all of the available components, examples, and demos for your target silicon. In addition to the MCUXpresso IDE, support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

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

Development tools

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

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

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

Supported development systems

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

Development boards	MCU devices
KW47-LOC	KW47B42ZB7AFTA

MCUXpresso SDK release package

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

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

for each supported toolchain. The startup code efficiently transfers the code execution to the `main()` function.

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

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

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

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

Main Features supported:

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

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

Bluetooth LE Controller

- Main features supported:
 - Peripheral Role
 - Central Role
 - Multiple PHYs (1 Mbps, 2 Mbps, Coded PHY)
 - Asymmetric Connections
 - Public/Random/Static Addresses
 - Network/Device Privacy Modes

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

- Randomized Resolvable Private Address (RPA).

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

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

Wireless Localization Localization

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

Supported services

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

Supported platform

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

Advanced features supported on platforms

KW45_MCXW71

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

KW47_MCXW72

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

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

TF-M Trusted Firmware - M Library

PSA Test Suite Arm Platform Security Architecture Test Suite

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

secure_storage secure_storage

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

Multicore Multicore Software Development Kit

NXP IoT Agent NXP IoT Agent

mbedTLS mbedtls SSL/TLS library v3.x

mbedTLS mbedtls SSL/TLS library v2.x

LittleFS LittleFS filesystem stack

LIN Stack LIN Stack middleware

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

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

What is new

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

• Bluetooth LE Host Stack and Applications

Added

- CCC and IoT Channel Sounding Localization applications on the GitHub repository
- Local average and remote average RSSI values to the CS measurement report
- The “-Os” optimization flag to the ARMGCC release configuration for the NCP applications

Improved

- Enabled low power support in the ‘loc_reader_host’ application

Fixed

- ‘loc_reader_host’ application event set issue
- Missing handler for Version2 of the Set RPA Timeout command
- Details can be found in github repository [nxp-mcuxpresso/mcuxsdk-middleware-bluetooth-host/CHANGELOG.md](https://github.com/nxp-mcuxpresso/mcuxsdk-middleware-bluetooth-host/CHANGELOG.md).

• Bluetooth LE controller

- Channel sounding updates:
 - * Fixed HCI_LE_CS_Config_Complete event which was not sent when rejecting the peripheral's request in case of colliding procedure.
 - * Adjusted RTT vs temperature compensation.
- Periodic Advertising updates:
 - * Fixed PerAdvSync skip scheduling.
 - * Improved PAwR Responses (TX & RX) instant.
 - * Fixed calculation of the count parameters in Periodic Advertising Subevent Data Request.
- Fixed a corner case where LL_CONNECTION_PARAM_REQ is immediately followed by LL_TERMINATE_IND.
- **Transceiver drivers (XCVR)**
 - Added support for Bluetooth LE Channel Sounding.
 - Added API to control PA ramp type and duration.

- **Connectivity framework**

Major Changes - [wireless_mcu][wireless_nbu] Replaced interrupt masking macros with static inline functions PLATFORM_SetInterruptMask() and PLATFORM_ClearInterruptMask() to ensure consistent BASEPRI value handling across all compilers. This addresses compiler-dependent behavior issues with the previous macro implementation. - [wireless_mcu] Added BASEPRI-based interrupt masking in PLATFORM_RemoteActiveRel() to allow high-priority IRQs while ensuring only IMU0 or thread context can call this function. - [wireless_mcu] Introduced gPlatformUseHwParameter_d compile flag to allow builds without HwParameter section. When undefined or set to 0, crystal trimming functions conditionally access HwParameters only when required. **Minor Changes** - [kw47_mcxw72] Introduced platform-specific library for KW47/MCXW72 platforms. - [kw47_mcxw72] Support for additional KW47 phantoms including ZB2/ZB3/ZB6/ZB7 and Z83/Z96/Z97. - [kw47_mcxw72] Removed SH_MEM_TOTAL_SIZE override as it is now automatically calculated to match rpmsg-lite configuration. - [wireless_mcu] Set RL_BUFFER_PAYLOAD_SIZE to word-aligned value as expected by rpmsg-lite. - [wireless_mcu] Added system-generated HCI vendor events capability for debug and diagnostic purposes. - [DBG] Added debug structure transmission over HCI vendor events with NBUDBG_ConfigureHciVendorEvent() API to enable/disable the feature. - [DBG] Added debug structure versioning and logging buffer address/size information to debug structure. - [DBG] Removed 15.4 region from debug structure as not supported on KW47. - [DBG] Added stack overflow detection for armv8_m_mainline architecture. - [Platform] Simplified enablement of reset features via pin detection - Automatically selects gUseResetByLvdForce_c when gAppForceLvdResetOnResetPinDet_d is enabled. - Automatically select gUseResetByDeepPowerDown_c when gAppForceDeepPowerDownResetOnResetPinDet_d is enabled. - [RNG] Replaced gRngHasSecLibDependency_d compilation switch with gRngUseSecLib_d. **Bug fixes** - [wireless_mcu] Fixed race condition in PLATFORM_RemoteActiveRel() by adding verification loop to confirm NBU core execution before releasing power domain. - [wireless_mcu] Added instruction synchronization barrier (__ISB()) after interrupt re-enable in PLATFORM_RemoteActiveRel() to ensure pending interrupts execute between critical sections. - [wireless_mcu] Fixed external IO voltage isolation issue during low-power initialization - isolation is now cleared at init to ensure proper behavior. - [wireless_mcu] Replaced spin-wait loops with event-based synchronization in NBU communication APIs. Added mutex protection to PLATFORM_NbuApiReq() and PLATFORM_GetNbuInfo() to prevent race conditions and deadlocks when multiple tasks call these APIs concurrently. - [wireless_mcu] Fixed OSA bare metal event race condition in ICS where auto-clear event feature could cause tasks to become permanently stuck. Disabled auto-clear feature in bare metal builds and manually clear event flags after OSA_EventWait() returns successfully. - [NVM] Fixed NvIdle() to

prevent looping for more operations than the queue size. - [NVS] Fixed blank check procedure to return false (non-blank) when checking a 0 length area. - [NVS] Made external and internal flash ports consistent. - [DBG] Fixed debug structure size and callback access issues - corrected memory placement overlap between reg_info and assert_info. - [MISRA] Various MISRA compliance fixes in NVM, HWParameter, LowPower, SecLib, Platform modules and IFR offset definitions.

Known issues

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

FRO6M Clock Stability Issue

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

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

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

Maximum file path length in Windows 7 operating system

The Windows 7 operating system imposes a 260-character maximum length for file paths. When installing the MCUXpresso SDK, place it in a directory close to the root to prevent file paths from exceeding the maximum character length specified by the Windows operating system. The recommended location is the C:\nxp folder.

Only FreeRTOS is tested for RTOS support

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

Bluetooth LE

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

Bluetooth LE controller:

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

Periodic Advertising with Responses (PAWR):

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

KW45/MCXW71: No specific issues.

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

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

Other limitations

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

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

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

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

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version
-

CACHE LPCAC

[2.1.2]

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

[2.1.1]

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

[2.1.0]

- Bug Fixes
 - Deleted L1CACHE_EnableCodeCacheWriteBuffer function because of no enable bit in register CPC2.

[2.0.0]

- Initial version.
-

CCM32K

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

CLOCK

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

- Bug fix:
 - Updated the `SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY` for Core1.

[2.2.1]

- Bug fix:
 - Fixed the `GPIO_PinInit` function may cause the hardfault in clock driver when pass the `GPIO_D` as the first parameter.

[2.2.0]

- Improvements
 - Added the support for TPM2.

[2.1.0]

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

[2.0.0]

- Fixed the MISRA issues.

[1.1.0]

- Some minor fixes.

[1.0.0]

- Initial version.
-

CMC**[2.4.4]**

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

[2.4.3]

- Improvements
 - Add timeout for while loop code.

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

COMMON**[2.6.3]**

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

[2.6.2]

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

[2.6.1]

- Improvements
 - Support Cortex M23.

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

- Improvements
 - Enable irq 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 `stdint.h` header file for MDK tool chain.
- New Features:
 - Added atomic modification macros.

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

CRC**[2.0.5]**

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

[2.0.4]

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

[2.0.3]

- Bug fix:
 - Fix MISRA issues.

[2.0.2]

- Bug fix:
 - Fix MISRA issues.

[2.0.1]

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

[2.0.0]

- Initial version.
-

EDMA (DMA3)

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.7]

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

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

- Added source/destination address alignment check.

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

ELEMU

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

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

EWM

[2.0.4]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed the hard fault in EWM_Deinit.

[2.0.0]

- Initial version.
-

FLASH

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

- New features
- Add support for MCX N series

[2.2.1]

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

[2.2.0]

- New Features
 - Add the support for MW30.

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

FLEXCAN

[2.14.5]

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

[2.14.4]

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

[2.14.3]

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

[2.14.2]

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

[2.14.1]

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

[2.14.0]

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

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

[2.13.1]

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

[2.13.0]

- Improvements
 - Support payload endianness selection feature.

[2.12.0]

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

[2.11.8]

- Improvements
 - Synchronize flexcan driver update on s32z platform.

[2.11.7]

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

[2.11.6]

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

[2.11.5]

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

[2.11.4]

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

[2.11.3]

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

[2.11.2]

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

[2.11.1]

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

[2.11.0]

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

[2.10.1]

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

[2.10.0]

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

[2.9.2]

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

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

[2.9.1]

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

[2.9.0]

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

[2.8.7]

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

[2.8.6]

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

[2.8.5]

- Improvements
 - Make driver aarch64 compatible.

[2.8.4]

- Bug Fixes
 - Fixed FlexCan_Errata_6032 to disable all interrupts.

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.1]

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

[2.7.0]

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

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

[2.6.2]

- Improvements
 - Add CANFD frame data length enumeration.

[2.6.1]

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

[2.6.0]

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

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.7, rule-10.8, rule-11.8, rule-12.2, rule-13.4, rule-14.4, rule-15.5, rule-15.6, rule-15.7, rule-16.4, rule-17.3, rule-5.8, rule-8.3, rule-8.5.
 - Fixed the issue that API FLEXCAN_SetFDRxMbConfig lacks inactive message buff.
 - Fixed the issue of Pa082 warning.
 - Fixed the issue of dead lock in the function of interruption handler.
 - Fixed the issue of Legacy Rx Fifo EDMA transfer data fail in evkmimxrt1060 and evk-mimxrt1064.
 - Fixed the issue of setting CANFD Bit Rate Switch.
 - Fixed the issue of operating unknown pointer risk.
 - * when used the pointer "handle->mbFrameBuf[mbIdx]" to update the timestamp in a short-live TX frame, the frame pointer became as unknown, the action of operating it would result in program stack destroyed.
 - Added assert to check current CAN clock source affected by other clock gates in current device.
 - * In some chips, CAN clock sources could be selected by CCM. But for some clock sources affected by other clock gates, if user insisted on using that clock source, they had to open these gates at the same time. However, they should take into consideration the power consumption issue at system level. In RT10xx chips, CAN clock source 2 was affected by the clock gate of lpuart1. ERRATA ID: (ERR050235 in CCM).

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

[2.4.0]

- Bug Fixes
 - MISRA C-2012 issue check.

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

FLEXCAN_EDMA**[2.12.1]**

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

[2.12.0]

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

[2.11.7]

- Refer FLEXCAN driver change log 2.7.0 to 2.11.7
-

FLEXIO

[2.3.0]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Improvements
 - Added doxygen index parameter comment in FLEXIO_SetClockMode.

[2.2.0]

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

[2.1.0]

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

[2.0.4]

- Bug Fixes
 - Fixed MISRA 8.4 issues.

[2.0.3]

- Bug Fixes
 - Fixed MISRA 10.4 issues.

[2.0.2]

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

[2.0.1]

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

FLEXIO_I2C**[2.6.2]**

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues

[2.6.0]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_I2S

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_SPI**[2.4.3]**

- Improvements
 - Make SPI_RETRY_TIMES configurable by CONFIG_SPI_RETRY_TIMES.

[2.4.2]

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

[2.4.1]

- Bug Fixes
 - Fixed coverity issues

[2.4.0]

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

[2.3.5]

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

[2.3.4]

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

[2.3.3]

- Bugfixes
 - Fixed cs-continuous mode.

[2.3.2]

- Improvements
 - Changed FLEXIO_SPI_DUMMYDATA to 0x00.

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

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

FLEXIO_UART

[2.6.4]

- Improvements
 - Make UART_RETRY_TIMES configurable by CONFIG_UART_RETRY_TIMES.

[2.6.3]

- Bug Fixes
 - Fixed coverity issues

[2.6.2]

- Bug Fixes
 - Fixed coverity issues

[2.6.1]

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

[2.6.0]

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

[2.5.1]

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

[2.5.0]

- Improvements
 - Added API FLEXIO_UART_FlushShifters to flush UART fifo.

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_UART_EDMA

[2.3.1]

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

[2.3.0]

- Refer FLEXIO_UART driver change log to 2.3.0
-

GPIO**[2.8.3]**

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

[2.8.2]

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

[2.8.1]

- Bug Fixes
 - Fixed CERT INT31-C issues.

[2.8.0]

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

[2.8.0]

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

[2.7.3]

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

[2.7.2]

- New Features
 - Support devices without PORT module.

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

I3C

[2.14.3]

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

[2.14.2]

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

[2.14.1]

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

[2.14.0]

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

[2.13.1]

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

[2.13.0]

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

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

[2.12.0]

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

[2.11.1]

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

[2.11.0]

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

[2.10.8]

- Improvements
 - Support more instances.

[2.10.7]

- Improvements
 - Fixed the potential compile warning.

[2.10.6]

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

[2.10.5]

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

[2.10.4]

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

[2.10.3]

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

[2.10.2]

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

[2.10.1]

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

[2.10.0]

- New features
 - Added I3C extended IBI data support.

[2.9.0]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.4]

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

[2.5.3]

- Updated driver for new feature `FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH` and `FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND`.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

- Bug Fixes
 - Fixed `kI3C_SlaveMatchedFlag` interrupt is not properly handled in `I3C_SlaveTransferHandleIRQ` when it comes together with interrupt `kI3C_SlaveBusStartFlag`.
 - Fixed the inaccurate I2C baudrate calculation in `I3C_MasterSetBaudRate`.
 - Added new API `I3C_MasterGetIBIRules` to get registered IBI rules.
 - Added new variable `isReadTerm` in `struct_i3c_master_handle` for transfer state routine to check if `MCTRL.RDTERM` is configured for read transfer.
 - Changed to emit Auto IBI in transfer state routine for slave start flag assertion.
 - Fixed the slave `maxWriteLength` and `maxReadLength` does not be configured into `SMAXLIMITS` register issue.
 - Fixed incorrect state for IBI in I3C master interrupt transfer IRQ handle routine.
 - Added `isHotJoin` in `i3c_slave_config_t` to request hot-join event during slave init.

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

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

[2.2.0]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

IMU

[2.2.1]

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

[2.2.0]

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

- Added IMU_ERR_TIMEOUT error code for timeout conditions.

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

LIN

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPADC**[2.9.5]**

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

[2.9.4]

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

[2.9.3]

- Improvements
 - Add timeout for while loop code.

[2.9.2]

- Improvements
 - Fixed CERT-C issues.

[2.9.1]

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

[2.9.0]

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

[2.8.4]

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

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.2]

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

[2.7.1]

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

[2.7.0]

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

[2.6.2]

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

[2.6.1]

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

[2.6.0]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

- Improvements
 - Added missing items to enable trigger interrupts.

[2.4.0]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPCMP

[2.3.2]

- Improvements
 - Fixed LPCMP CERT-C issues.

[2.3.1]

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

[2.3.0]

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

[2.2.0]

- Improvements
 - Change `FSL_FEATURE_LPCMP_HAS_NO_CCR0_CMP_STOP_EN` to `FSL_FEATURE_LPCMP_HAS_CCR0_CMP_STOP_EN`.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPI2C**[2.6.3]**

- Bug Fixes
 - Fixed static analysis identified issues.

[2.6.2]

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues.

[2.6.0]

- New Feature
 - Added common IRQ handler entry LPI2C_DriverIRQHandler.

[2.5.7]

- Improvements
 - Added support for separated IRQ handlers.

[2.5.6]

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

[2.5.5]

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

[2.5.4]

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

[2.5.3]

- Improvements
 - Added handler for LPI2C7 and LPI2C8.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.1]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.0]

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

[2.1.12]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.1.11]

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

[2.1.10]

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

[2.1.9]

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

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

LPI2C_EDMA

[2.4.6]

- Bug Fixes
 - Fixed static analysis identified issues.

[2.4.5]

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

[2.4.4]

- Improvements
 - Added support for 2KB data transfer

[2.4.3]

- Improvements
 - Added support for separated IRQ handlers.

[2.4.2]

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

[2.4.1]

- Refer LPI2C driver change log 2.0.0 to 2.4.1
-

LPIT

[2.1.3]

- Bug Fixes
 - Fixed doxygen generation warnings.

[2.1.2]

- Bug Fixes
 - Fix CERT INT31-C issues.

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPSPi**[2.7.4]**

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

[2.7.3]

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

[2.7.2]

- Bug Fixes
 - Fixed coverity issues.

[2.7.1]

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

[2.7.0]

- New Feature
 - Added common IRQ handler entry LPSPi_DriverIRQHandler.

[2.6.10]

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

[2.6.9]

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

[2.6.8]

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

[2.6.7]

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

[2.6.6]

- Bug Fixes
 - Added LPSPi register init in LPSPi_MasterInit incase of LPSPi register exist.

[2.6.5]

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

[2.6.4]

- Bug Fixes
 - Added LPSPi6_DriverIRQHandler for LPSPi6 instance.

[2.6.3]

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

[2.6.2]

- Bug Fixes
 - Disabled lpspi before LPSPi_MasterSetBaudRate incase of LPSPi opened.

[2.6.1]

- Bug Fixes
 - Fixed return value while calling LPSPi_WaitTxFifoEmpty in function LPSPi_MasterTransferNonBlocking.

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.1]

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

[2.3.0]

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

[2.2.1]

- Bug Fixes
 - Fixed bug in LPSPI_SetPCSContinuous when disabling PCS continuous mode.

[2.2.0]

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

[2.1.0]

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

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

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

[2.0.0]

- Initial version.
-

LPSPI_EDMA

[2.4.9]

- Improvements
 - Removed unused code from LPSPI_SeparateEdmaReadData().

[2.4.8]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

- Improvements
 - Add the `TXMSK` wait after `TCR` setting.

[2.4.0]

- Improvements
 - Separated `LPSPI_MasterTransferEDMA` functions to `LP-SPI_MasterTransferPrepareEDMA` and `LPSPI_MasterTransferEDMALite` to optimize the process of transfer.
-

LPTMR**[2.2.1]**

- Bug Fixes
 - Fix CERT INT31-C issues.

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPUART

[2.10.0]

- New Feature
 - Added support to configure RTS watermark.

[2.9.4]

- Improvements
 - Merged duplicate code.

[2.9.3]

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

[2.9.2]

- Bug Fixes
 - Fixed coverity issues.

[2.9.1]

- Bug Fixes
 - Fixed coverity issues.

[2.9.0]

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

[2.8.3]

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

[2.8.2]

- Bug Fix
 - Fixed the bug that LPUART_TransferEnable16Bit 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 has a negative effect on other IPs that are using the interrupt.

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - Update transactional APIs.
-

LPUART_EDMA

[2.4.0]

- Refer LPUART driver change log 2.1.0 to 2.4.0
-

LTC

[2.0.18]

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

[2.0.17]

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

[2.0.16]

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

[2.0.15]

- Bug fix:
 - Fix MISRA-2012 issues.

[2.0.14]

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

[2.0.13]

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

[2.0.12]

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

[2.0.11]

- Bug fix:
 - Fix MISRA-2012 issues.

[2.0.10]

- Bug fix:
 - Fix MISRA-2012 issues.

[2.0.9]

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

[2.0.8]

- Bug fix:
 - Fixed coverity issues introduced in 2.0.7

[2.0.7]

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

[2.0.6]

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

[2.0.5]

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

[2.0.4]

- Improvements:
 - Constant `LTC_PKHA_CompareBigNum()` processing time.

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

MCM

[2.2.0]

- Improvements
 - Support platforms with less features.

[2.1.0]

- Others
 - Remove byteID from mcm_lmem_fault_attribute_t for document update.

[2.0.0]

- Initial version.
-

MSCM

[2.0.0]

- Initial version.
-

MU

[2.8.1]

- Bug Fixes
 - Avoid incorrect MU_BUSY_POLL_COUNT macro use.

[2.8.0]

- New Features
 - Added MU1_BUSY_POLL_COUNT parameter to prevent infinite polling loops in MU operations.
 - Added timeout mechanism to all polling loops in MU driver code.
- Improvements
 - Updated function signatures to return status codes for better error handling:
 - * Changed MU_ResetBothSides to return status_t instead of void
 - * Updated MU_SendMsg to return status_t for timeout indication
 - * Added new function MU_ReceiveMsgTimeout() to include timeout mechanism.
 - Enhanced documentation across all functions to clarify timeout behavior and return values.

[2.7.0]

- New Features
 - Added API MU_GetRxStatusFlags.

[2.6.0]

- New Features
 - Added API MU_GetInterruptsPending.

[2.5.1]

- Bug Fixes
 - Fixed the bug that MU_TriggerGeneralPurposeInterrupts and MU_TriggerInterrupts may trigger previous triggered general purpose interrupts again by mistake.

[2.5.0]

- New Features
 - Supported more than 4 general purpose interrupts.
 - Added separate APIs for general purpose interrupts.

[2.4.0]

- Improvements
 - Supported the case that some features only available with specific instances. These features include Hardware Reset, Boot Peer Core, Hold Reset. When using the features with instances which don't support them, driver will report error.

[2.3.3]

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

[2.3.2]

- Improvements
 - Supported platforms which don't have CCR0[RSTH], CCR0[CLKE], CCR0[HR], CCR0[HRM].

[2.3.1]

- Bug Fixes
 - Fixed build error for platforms which have CCR0[RSTH], but no CCR0[NMI].

[2.3.0]

- New features
 - Added support for i.MX RT7xx.

[2.2.1]

- Bug Fixes
 - Fixed issue that MU_GetInstance() is defined but never used.

[2.2.0]

- New features
 - Added support for i.MX RT118x.
- Bug Fixes
 - Fixed general purpose interrupt bug.
- Other Changes
 - Change _mu_interrupt_trigger item value.

[2.1.2]

- Bug Fixes
 - Fixed bug that general purpose interrupt can't be configured.

[2.1.1]

- Bug Fixes
 - Fixed MISRA C-2012 issues.

[2.1.0]

- Improvements
 - Added new enum mu_msg_reg_index_t.

[2.0.0]

- Initial version.
-

PORT

[2.5.1]

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

[2.5.0]

- Bug Fixes
 - Correct the kPORT_MuxAsGpio for some platforms.

[2.4.1]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

- New Features
 - Added new api PORT_EnablePinDoubleDriveStrength.

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

RTC

[2.4.0]

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

[2.3.3]

- Bug Fixes
 - Fix RTC_GetDatetime function validating datetime issue.

[2.3.2]

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

[2.3.1]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.3.0]

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

[2.2.2]

- Improvements
 - Refine _rtc_interrupt_enable order.

[2.2.1]

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

[2.2.0]

- Bug Fixes
 - Fixed MISRA C-2012 issue.
 - * Fixed rule contain: rule-17.7, rule-14.4, rule-10.4, rule-10.7, rule-10.1, rule-10.3.
 - Fixed central repository code formatting issue.

- Improvements
 - Added an API for enabling wakeup pin.

[2.1.0]

- Improvements
 - Added feature macro check for many features.

[2.0.0]

- Initial version.
-

SEMA42

[2.1.1]

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

[2.1.0]

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

[2.0.4]

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

[2.0.3]

- Improvements
 - Changed to implement SEMA42_Lock base on SEMA42_TryLock.

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

SFA

[2.1.4]

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

[2.1.3]

- Improvements
 - Add timeout for APIs with dfmea issues.

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

SMSCM**[2.0.0]**

- Initial version.
-

SPC**[2.8.0]**

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

[2.7.1]

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

[2.7.0]

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

[2.6.1]

- Improvements
 - Add timeout for while loop code.

[2.6.0]

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

[2.5.0]

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

[2.4.1]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.0]

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

- Fixed some issues in Bandgap control APIs.
- Fixed some issues in regulator active/lowpower mode configuration APIs.

[2.0.2]

- Bug Fixes
 - Fixed the doxygen issue by adding missed comments.

[2.0.1]

- Bug Fixes
 - The enumerator `spc_core_ldo_voltage_level_t` may differ for different devices, added the macro to separate different scenes.

[2.0.0]

- Initial version.
-

SYSPM

[2.3.1]

- Improvements
 - Add timeout for while loop.

[2.3.0]

- New Features
 - Supported instruction counter.

[2.2.0]

- New Features
 - Supported platforms which only have one monitor.
 - Supported platforms whose instruction counter can't be cleared.
 - Supported platforms whose counter can't be disabled.

[2.1.0]

- New Features
 - Added new APIs `SYSPM_Init` and `SYSPM_Deinit`.

[2.0.0]

- Initial version.
-

TPM

[2.4.1]

- Improvements
 - Add Coverage Justification for uncovered code.

[2.4.0]

- New Feature
 - Added while loop timeout for MOD CnV CnSC and SC register write sequence.
 - Change the return type from void to status_t for following API:
 - * TPM_DisableChannel
 - * TPM_EnableChannel
 - * TPM_SetupOutputCompare
 - * TPM_SetTimerPeriod
 - * TPM_StopTimer

[2.3.6]

- Bug Fixes
 - Fixed CERT INT30-C INT31-C issue for TPM_SetupDualEdgeCapture.

[2.3.5]

- New Feature
 - Added IRQ handler entry for TPM2.

[2.3.4]

- New Feature
 - Added common IRQ handler entry TPM_DriverIRQHandler.

[2.3.3]

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

[2.3.2]

- Bug Fixes
 - Fixed ERR008085 TPM writing the TPMx_MOD or TPMx_CnV registers more than once may fail when the timer is disabled.

[2.3.1]

- Bug Fixes
 - Fixed compilation error when macro `FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL` is 1.

[2.3.0]

- Improvements
 - Create callback feature for TPM match and timer overflow interrupts.

[2.2.4]

- Improvements
 - Add feature macros(`FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_EN`, `FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_SYNC`).

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Bug Fixes
 - Fixed CCM issue by splitting function from `TPM_SetupPwm()` function to reduce function complexity.
 - Fixed violations of MISRA C-2012 rule 17.7.

[2.2.0]

- Improvements
 - Added `TPM_SetChannelPolarity` to support select channel input/output polarity.
 - Added `TPM_EnableChannelExtTrigger` to support enable external trigger input to be used by channel.
 - Added `TPM_CalculateCounterClkDiv` to help calculates the counter clock prescaler.
 - Added `TPM_GetChannelValue` to support get TPM channel value.
 - Added new TPM configuration.
 - * `syncGlobalTimeBase`
 - * `extTriggerPolarity`
 - * `chnlPolarity`
 - Added new PWM signal configuration.

- * secPauseLevel

- Bug Fixes
 - Fixed TPM_SetupPwm can't configure 0% combined PWM issues.

[2.1.1]

- Improvements
 - Add feature macro for PWM pause level select feature.

[2.1.0]

- Improvements
 - Added TPM_EnableChannel and TPM_DisableChannel APIs.
 - Added new PWM signal configuration.
 - * pauseLevel - Support select output level when counter first enabled or paused.
 - * enableComplementary - Support enable/disable generate complementary PWM signal.
 - * deadTimeValue - Support deadtime insertion for each pair of channels in combined PWM mode.
- Bug Fixes
 - Fixed issues about channel MSnB:MSnA and ELSnB:ELSnA bit fields and CnV register change request acknowledgement. Writes to these bits are ignored when the interval between successive writes is less than the TPM clock period.

[2.0.8]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.4, 10.7 and 14.4.

[2.0.7]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.4 and 17.7.

[2.0.6]

- Bug Fixes
 - Fixed Out-of-bounds issue.

[2.0.5]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 10.6, 10.7

[2.0.4]

- Bug Fixes
 - Fixed ERR050050 in functions TPM_SetupPwm/TPM_UpdatePwmDutycycle. When TPM was configured in EPWM mode as PS = 0, the compare event was missed on the first reload/overflow after writing 1 to the CnV register.

[2.0.3]

- Bug Fixes
 - MISRA-2012 issue fixed.
 - * Fixed rules: rule-12.1, rule-17.7, rule-16.3, rule-14.4, rule-1.3, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-10.6, and rule-18.1.

[2.0.2]

- Bug Fixes
 - Fixed issues in functions TPM_SetupPwm/TPM_UpdateChnEdgeLevelSelect/TPM_SetupInputCapture/TPM_SetupOutputCompare/TPM_SetupDualEdgeCapture, wait acknowledgement when the channel is disabled.

[2.0.1]

- Bug Fixes
 - Fixed TPM_UpdateChnEdgeLevelSelect ACK wait issue.
 - Fixed the issue that TPM_SetupDualEdgeCapture could not set FILTER register.
 - Fixed TPM_UpdateChnEdgeLevelSelect ACK wait issue.

[2.0.0]

- Initial version.
-

TRDC**[2.3.1]**

- Improvements
 - Update APIs to check whether the memory access configuration can be updated.
 - Update APIs to mask the MRC address since only high 18 bits are valid.

[2.3.0]

- New Features
 - Added API TRDC_EnableProcessorDomainAssignment to disable/enable DAC.
- Bug Fixes
 - Fixed MISRA violation of missing function declaration.
 - Fixed wrong operation of domain mask in TRDC_MbcNseClearAll and TRDC_MrcDomainNseClear.

[2.2.1]

- Bug Fixes
 - Fixed MISRA violations of rule 10.3.

[2.2.0]

- New Features
 - Added new APIs to support SoC with more than one processor core.

[2.1.1]

- Bug Fixes
 - Fixed MISRA violations of rule 10.3, 10.6, 10.7 and 18.1.

[2.1.0]

- Improvements
 - Modified some of the addressing method according to the device header file's update.
 - Modified flash address configuration structure and default setting.
- New Features
 - Added API to set flash logic window array address.
- Bug Fixes
 - Fixed wrong return value of TRDC_GetFlashLogicalWindowPbase.
 - Fixed bug in TRDC_GetAndClearFirstSpecificDomainError that the error status is cleared by mistake before obtained by software.
 - Fixed MISRA violations of rule 10.3, 10.4, 10.6 and 10.8.

[2.0.0]

- Initial version.
-

TRGMUX

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.8.

[2.0.0]

- Initial version.
-

TSTMR

[2.1.0]

- New Features
 - Support configured clock frequency.
 - Add TSTMR_Init and TSTMR_init APIs.
- Improvements
 - Change TSTMR_DelayUs from static inline function to normal function.

[2.0.4]

- Bugfix
 - Fix MISRA C-2012 Rule 10.4 and 14.4 issues.

[2.0.3]

- Bugfix
 - Fix CERT INT30-C that Unsigned integer operation TSTMR_ReadTimeStamp(base) - startTime may wrap.

[2.0.2]

- Improvements
 - Support 24MHz clock source.
- Bugfix
 - Fix MISRA C-2012 Rule 10.4 issue.
 - Read of TSTMR HIGH must follow TSTMR LOW atomically: require masking interrupt around 2 LSB / MSB accesses.

[2.0.1]

- Bugfix
 - Restrict to read with 32-bit accesses only.
 - Restrict that TSTMR LOW read occurs first, followed by the TSTMR HIGH read.

[2.0.0]

- Initial version.
-

VBAT

[2.1.1]

- Improvements
 - Added timeout for while loop.

[2.1.0]

- Improvements
 - Both FROCFGA and LDOCFGA registers are set as internal register, removed corresponding interfaces.

[2.0.0]

- Initial version.
-

VREF

[2.4.1]

- Improvements
 - Improve doxygen documentation.

[2.4.0]

- Improvements
 - Support TEST_UNLOCK and TRIM0 registers read and write.

[2.3.0]

- Improvements
 - Add feature macros for compatibility with some platforms that don't have CSR[LPBG_BUF_EN] and UTRIM[TRIM2V1].

[2.2.2]

- Bug Fixes
 - Fixed typo in vref_buffer_mode_t enumerator.

[2.2.1]

- Improvements
 - Updated VREF driver code to reduce code redundancy.

[2.2.0]

- Improvements
 - Remove API VREF_SetTrimVal and VREF_GetTrimVal
 - Add new API VREF_SetVrefTrimVal, VREF_SetTrim21Val, VREF_GetVrefTrimVal and VREF_GetTrim21Val
 - Updated VREF driver code to conform to VREF IP architecture.

[2.1.0]

- Improvements
 - Remove API VREF_SetTrimVal and VREF_GetTrimVal
 - Add new API VREF_SetVrefTrimVal, VREF_SetTrim21Val, VREF_GetVrefTrimVal and VREF_GetTrim21Val
 - Updated VREF driver code to conform to VREF IP architecture.
- Bug Fixes
 - Fix issue which progeam stuck in VREF_Init

[2.0.0]

- Initial version.
-

WDOG32**[2.2.1]**

- Bug Fixes
 - Fix CERT INT31-C that the bool value shall be converted to unsigned int 0 or 1 then passed to registers.
 - Fix MISRA 2012 20.3 vilation.

[2.2.0]

- Improvements
 - Added while loop timeout config value for WDOG32 reconfiguration and unlock sequence.
 - Change the return type of WDOG32_Init, WDOG32_Deinit and WDOG32_Unlock from void to status_t.

[2.1.0]

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

[2.0.4]

- Improvements
 - To ensure that the reconfiguration is inside 128 bus clocks unlock window, put all re-configuration APIs in quick access code section.

[2.0.3]

- Bug Fixes
 - Fixed the noncompliance issue of the reference document.
 - * Waited until for new configuration to take effect by checking the RCS bit field.
 - * Waited until for registers to be unlocked by checking the ULK bit field.
- Improvements
 - Added 128 bus clocks delay ensures a smooth transition before restarting the counter with the new configuration when there is no RCS status bit.

[2.0.2]

- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fixed rules, containing: rule-10.3, rule-14.4, rule-15.5.
 - Fixed the issue of the inseparable process interrupted by other interrupt source.
 - * WDOG32_Refresh

[2.0.1]

- Bug Fixes
 - WDOG must be configured within its configuration time period.
 - * Added WDOG32_Init API to quick access section.
 - * Defined register variable in WDOG32_Init API.

[2.0.0]

- Initial version.
-

WUU

[2.4.1]

- Improvements
 - The semantics of kWUU_FilterActiveDSPD and kWUU_ExternalPinActiveDSPD are not clear enough, because on some platforms it's not 'DSPD' (deep sleep, power down) but PDDPD (power down, deep power down). We deprecate them and will use the more generic kWUU_FilterActiveLowLeakage and kWUU_ExternalPinActiveLowLeakage in the future.

[2.4.0]

- New Features
 - Added WUU_ClearExternalWakeupPinsConfig() to clear settings of PDC and PE register.

[2.3.0]

- New Features
 - Added `WUU_ClearInternalWakeUpModulesConfig()` to clear settings of DM and ME register.

[2.2.1]

- Bug Fixes
 - Fixed `WUU_SetPinFilterConfig()` unable to set edge detection of pin filter config.
 - Fixed wrong macro used in `WUU_GetPinFilterFlag()` function.

[2.2.0]

- New Features
 - Added the `WUU_GetExternalWakeupPinFlag()` and `WUU_ClearExternalWakeupPinFlag()` function .

[2.1.0]

- New Features
 - Added the `WUU_GetModuleInterruptFlag()` function to support the devices that equipped MF register.

[2.0.0]

- Initial version.
-

1.6 Driver API Reference Manual

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

[KW47B42ZB7](#)

1.7 Middleware Documentation

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

1.7.1 Wireless Bluetooth LE host stack and applications

[examples__wireless_examples__bluetooth_docs](#)

1.7.2 Wireless Connectivity Framework

[framework](#)

1.7.3 Multicore

Multicore SDK

1.7.4 FreeMASTER

freemaster

1.7.5 FreeRTOS

FreeRTOS

Chapter 2

KW47B42ZB7

2.1 CACHE: LPCAC CACHE Memory Controller

static inline void L1CACHE_EnableCodeCache(void)

Enables the processor code bus cache.

static inline void L1CACHE_DisableCodeCache(void)

Disables the processor code bus cache.

static inline void L1CACHE_InvalidateCodeCache(void)

Invalidates the processor code bus cache.

void L1CACHE_InvalidateICacheByRange(uint32_t address, uint32_t size_byte)

Invalidates L1 instrument cache by range.

Parameters

- address – The start address of the memory to be invalidated.
- size_byte – The memory size.

static inline void L1CACHE_InvalidateDCacheByRange(uint32_t address, uint32_t size_byte)

Invalidates L1 data cache by range.

Parameters

- address – The start address of the memory to be invalidated.
- size_byte – The memory size.

static inline void L1CACHE_CleanDCacheByRange(uint32_t address, uint32_t size_byte)

Cleans L1 data cache by range.

The cache is write through mode, so there is nothing to do with the cache flush/clean operation.

Parameters

- address – The start address of the memory to be cleaned.
- size_byte – The memory size.

static inline void L1CACHE_CleanInvalidateDCacheByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates L1 data cache by range.

Parameters

- address – The start address of the memory to be clean and invalidated.

- size_byte – The memory size.

static inline void ICACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)

Invalidates instruction cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated.

static inline void DCACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)

Invalidates data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated.

static inline void DCACHE_CleanByRange(uint32_t address, uint32_t size_byte)

Clean data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be cleaned.

static inline void DCACHE_CleanInvalidateByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be Cleaned and Invalidated.

FSL_CACHE_DRIVER_VERSION

cache driver version 2.1.2.

2.2 CCM32K: 32kHz Clock Control Module

void CCM32K_Enable32kFro(CCM32K_Type *base, bool enable)

Enable/Disable 32kHz free-running oscillator.

Note: There is a start up time before clocks are output from the FRO.

Note: To enable FRO32k and set it as 32kHz clock source please follow steps:

```
CCM32K_Enable32kFro(base, true);           //Enable FRO analog oscillator.
CCM32K_DisableCLKOutToPeripherals(base, mask); //Disable clock out.
CCM32K_SelectClockSource(base, kCCM32K_ClockSourceSelectFro32k); //Select FRO32k as clock_
↪source.
while(CCM32K_GetStatus(base) != kCCM32K_32kFroActiveStatusFlag); //Check FOR32k is active_
↪and in used.
CCM32K_EnableCLKOutToPeripherals(base, mask); //Enable clock out if needed.
```

Parameters

- base – CCM32K peripheral base address.

- `enable` – Boolean value to enable or disable the 32kHz free-running oscillator. `true` — Enable 32kHz free-running oscillator. `false` — Disable 32kHz free-running oscillator.

static inline void CCM32K_Lock32kFroWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the FRO32K_CTRL register until a POR occurs.

Parameters

- `base` – CCM32K peripheral base address.

static inline uint16_t CCM32K_Get32kFroTrimValue(CCM32K_Type *base)

Get frequency trim value of 32kHz free-running oscillator.

Parameters

- `base` – CCM32K peripheral base address.

Returns

The current trim value.

void CCM32K_Set32kFroTrimValue(CCM32K_Type *base, uint16_t trimValue)

Set the frequency trim value of 32kHz free-running oscillator by software.

Note: The frequency is decreased monotonically when the trimValue is changed progressively from 0x0U to 0x7FFU.

Note: If the FRO32 is enabled before invoking this function, then in this function the FRO32 will be disabled, after updating trim value the FRO32 will be re-enabled.

Parameters

- `base` – CCM32K peripheral base address.
- `trimValue` – The frequency trim value.

static inline void CCM32K_Disable32kFroIFRLoad(CCM32K_Type *base, bool disable)

Disable/Enable the function of setting 32kHz free-running oscillator trim value when IFR value gets loaded in the SOC.

Parameters

- `base` – CCM32K peripheral base address.
- `disable` – Boolean value to disable or enable IFR loading function. `true` — Disable IFR loading function. `false` — Enable IFR loading function.

static inline void CCM32K_Lock32kFroTrimWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the FRO32K_TRIM register until a POR occurs.

Parameters

- `base` – CCM32K peripheral base address.

void CCM32K_Set32kOscConfig(CCM32K_Type *base, *ccm32k_osc_mode_t* mode, const *ccm32k_osc_config_t* *config)

Config 32k Crystal Oscillator.

Note: When the mode selected as `kCCM32K_Disable32kHzCrystalOsc` or `kCCM32K_Bypass32kHzCrystalOsc` the parameter config is useless, so it can be set as

“NULL”.

Note: To enable OSC32K and select it as clock source of 32kHz please follow steps:

```
CCM32K_Set32kOscConfig(base, kCCM32K_Enable32kHzCrystalOsc, config); //Enable OSC32k and
↪set config.
while((CCM32K_GetStatus(base) & kCCM32K_32kOscReadyStatusFlag) == 0UL); //Check if
↪OSC32K is stable.
CCM32K_DisableCLKOutToPeripherals(base, mask); //Disable clock out.
CCM32K_SelectClockSource(base, kCCM32K_ClockSourceSelectOsc32k); //Select OSC32k as clock
↪source.
while((CCM32K_GetStatus(base) & kCCM32K_32kOscActiveStatusFlag) == 0UL); //Check if
↪OSC32K is used as clock source.
CCM32K_EnableCLKOutToPeripherals(base, mask); //Enable clock out.
```

Parameters

- base – CCM32K peripheral base address.
- mode – The mode of 32k crystal oscillator.
- config – The pointer to the structure ccm32k_osc_config_t.

static inline void CCM32K_Lock32kOscWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the OSC32K_CTRL register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

void CCM32K_EnableClockMonitor(CCM32K_Type *base, bool enable)

Enable/disable clock monitor.

Parameters

- base – CCM32K peripheral base address.
- enable – Used to enable/disable clock monitor.
 - **turn** Enable clock monitor.
 - **false** Disable clock monitor.

```
static inline void CCM32K_SetClockMonitorFreqTrimValue(CCM32K_Type *base,
                                                         ccm32k_clock_monitor_freq_trim_value_t
                                                         trimValue)
```

Set clock monitor frequency trim value.

Parameters

- base – CCM32K peripheral base address.
- trimValue – Clock minitor frequency trim value, please refer to ccm32k_clock_monitor_freq_trim_value_t.

```
static inline void CCM32K_SetClockMonitorDivideTrimValue(CCM32K_Type *base,
                                                          ccm32k_clock_monitor_divide_trim_value_t
                                                          trimValue)
```

Set clock monitor divide trim value.

Parameters

- base – CCM32K peripheral base address.

- trimValue – Clock minitor divide trim value, please refer to `ccm32k_clock_monitor_divide_trim_value_t`.

```
void CCM32K_SetClockMonitorConfig(CCM32K_Type *base, const
                                ccm32k_clock_monitor_config_t *config)
```

Config clock monitor one time, including frequency trim value, divide trim value.

Parameters

- base – CCM32K peripheral base address.
- config – Pointer to `ccm32k_clock_monitor_config_t` structure.

```
static inline void CCM32K_LockClockMonitorWriteAccess(CCM32K_Type *base)
```

Lock all further write accesses to the CLKMON_CTRL register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

```
static inline void CCM32K_EnableCLKOutToPeripherals(CCM32K_Type *base, uint8_t
                                                    peripheralMask)
```

Enable 32kHz clock output to selected peripherals.

Parameters

- base – CCM32K peripheral base address.
- peripheralMask – The mask of peripherals to enable 32kHz clock output, should be the OR'ed value of `ccm32k_clock_output_peripheral_t`.

```
static inline void CCM32K_DisableCLKOutToPeripherals(CCM32K_Type *base, uint8_t
                                                    peripheralMask)
```

Disable 32kHz clock output to selected peripherals.

Parameters

- base – CCM32K peripheral base address.
- peripheralMask – The mask of peripherals to disable 32kHz clock output, should be the OR'ed value of `ccm32k_clock_output_peripheral_t`.

```
static inline void CCM32K_SelectClockSource(CCM32K_Type *base,
                                            ccm32k_clock_source_select_t clockSource)
```

Select CCM32K module's clock source which will be provide to the device.

Parameters

- base – CCM32K peripheral base address.
- clockSource – Used to select clock source, please refer to `ccm32k_clock_source_select_t` for details.

```
static inline void CCM32K_LockClockGateWriteAccess(CCM32K_Type *base)
```

Lock all further write access to the CGC32K register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

```
static inline uint32_t CCM32K_GetStatusFlag(CCM32K_Type *base)
```

Get the status flag.

Parameters

- base – CCM32K peripheral base address.

Returns

The status flag of the current node. The enumerator of status flags have been provided, please see the Enumerations title for details.

`ccm32k_state_t` CCM32K_GetCurrentState(CCM32K_Type *base)

Get current state.

Parameters

- base – CCM32K peripheral base address.

Returns

The CCM32K's current state, please refer to `ccm32k_state_t` for details.

`ccm32k_clock_source_t` CCM32K_GetClockSource(CCM32K_Type *base)

Return current clock source.

Parameters

- base – CCM32K peripheral base address.

Return values

- `kCCM32K_ClockSourceNone` – The none clock source is selected.
- `kCCM32K_ClockSource32kFro` – 32kHz free-running oscillator is selected as clock source.
- `kCCM32K_ClockSource32kOsc` – 32kHz crystal oscillator is selected as clock source..

FSL_CCM32K_DRIVER_VERSION

CCM32K driver version 2.2.1.

enum `_ccm32k_osc_xtal_cap`

The enumerator of internal capacitance of OSC's XTAL pin.

Values:

enumerator `kCCM32K_OscXtal0pFCap`

The internal capacitance for XTAL pin is 0pF.

enumerator `kCCM32K_OscXtal2pFCap`

The internal capacitance for XTAL pin is 2pF.

enumerator `kCCM32K_OscXtal4pFCap`

The internal capacitance for XTAL pin is 4pF.

enumerator `kCCM32K_OscXtal6pFCap`

The internal capacitance for XTAL pin is 6pF.

enumerator `kCCM32K_OscXtal8pFCap`

The internal capacitance for XTAL pin is 8pF.

enumerator `kCCM32K_OscXtal10pFCap`

The internal capacitance for XTAL pin is 10pF.

enumerator `kCCM32K_OscXtal12pFCap`

The internal capacitance for XTAL pin is 12pF.

enumerator `kCCM32K_OscXtal14pFCap`

The internal capacitance for XTAL pin is 14pF.

enumerator `kCCM32K_OscXtal16pFCap`

The internal capacitance for XTAL pin is 16pF.

enumerator kCCM32K_OscXtal18pFCap

The internal capacitance for XTAL pin is 18pF.

enumerator kCCM32K_OscXtal20pFCap

The internal capacitance for XTAL pin is 20pF.

enumerator kCCM32K_OscXtal22pFCap

The internal capacitance for XTAL pin is 22pF.

enumerator kCCM32K_OscXtal24pFCap

The internal capacitance for XTAL pin is 24pF.

enumerator kCCM32K_OscXtal26pFCap

The internal capacitance for XTAL pin is 26pF.

enumerator kCCM32K_OscXtal28pFCap

The internal capacitance for XTAL pin is 28pF.

enumerator kCCM32K_OscXtal30pFCap

The internal capacitance for XTAL pin is 30pF.

enum _ccm32k_osc_extal_cap

The enumerator of internal capacitance of OSC's EXTAL pin.

Values:

enumerator kCCM32K_OscExtal0pFCap

The internal capacitance for EXTAL pin is 0pF.

enumerator kCCM32K_OscExtal2pFCap

The internal capacitance for EXTAL pin is 2pF.

enumerator kCCM32K_OscExtal4pFCap

The internal capacitance for EXTAL pin is 4pF.

enumerator kCCM32K_OscExtal6pFCap

The internal capacitance for EXTAL pin is 6pF.

enumerator kCCM32K_OscExtal8pFCap

The internal capacitance for EXTAL pin is 8pF.

enumerator kCCM32K_OscExtal10pFCap

The internal capacitance for EXTAL pin is 10pF.

enumerator kCCM32K_OscExtal12pFCap

The internal capacitance for EXTAL pin is 12pF.

enumerator kCCM32K_OscExtal14pFCap

The internal capacitance for EXTAL pin is 14pF.

enumerator kCCM32K_OscExtal16pFCap

The internal capacitance for EXTAL pin is 16pF.

enumerator kCCM32K_OscExtal18pFCap

The internal capacitance for EXTAL pin is 18pF.

enumerator kCCM32K_OscExtal20pFCap

The internal capacitance for EXTAL pin is 20pF.

enumerator kCCM32K_OscExtal22pFCap

The internal capacitance for EXTAL pin is 22pF.

enumerator kCCM32K_OscExtal24pFCap

The internal capacitance for EXTAL pin is 24pF.

enumerator kCCM32K_OscExtal26pFCap

The internal capacitance for EXTAL pin is 26pF.

enumerator kCCM32K_OscExtal28pFCap

The internal capacitance for EXTAL pin is 28pF.

enumerator kCCM32K_OscExtal30pFCap

The internal capacitance for EXTAL pin is 30pF.

enum _ccm32k_osc_fine_adjustment_value

The enumerator of osc amplifier gain fine adjustment. Changes the oscillator amplitude by modifying the automatic gain control (AGC).

Values:

enumerator kCCM32K_OscFineAdjustmentRange0

enum _ccm32k_osc_coarse_adjustment_value

The enumerator of osc amplifier coarse fine adjustment. Tunes the internal transconductance (gm) by increasing the current.

Values:

enumerator kCCM32K_OscCoarseAdjustmentRange0

enumerator kCCM32K_OscCoarseAdjustmentRange1

enumerator kCCM32K_OscCoarseAdjustmentRange2

enumerator kCCM32K_OscCoarseAdjustmentRange3

enum _ccm32k_osc_mode

The enumerator of 32kHz oscillator.

Values:

enumerator kCCM32K_Disable32kHzCrystalOsc

Disable 32kHz Crystal Oscillator.

enumerator kCCM32K_Enable32kHzCrystalOsc

Enable 32kHz Crystal Oscillator.

enumerator kCCM32K_Bypass32kHzCrystalOsc

Bypass 32kHz Crystal Oscillator, use the 32kHz Oscillator or external 32kHz clock.

The enumerator of CCM32K status flag.

Values:

enumerator kCCM32K_32kOscReadyStatusFlag

Indicates the 32kHz crystal oscillator is stable.

enumerator kCCM32K_32kOscActiveStatusFlag

Indicates the 32kHz crystal oscillator is active and in use.

enumerator kCCM32K_32kFroActiveStatusFlag

Indicates the 32kHz free running oscillator is active and in use.

enumerator kCCM32K_ClockDetectStatusFlag

Indicates the clock monitor has detected an error.

enum _ccm32k_state

The enumerator of module state.

Values:

enumerator kCCM32K_Both32kFro32kOscDisabled

Indicates both 32kHz free running oscillator and 32kHz crystal oscillator are disabled.

enumerator kCCM32K_Only32kFroEnabled

Indicates only 32kHz free running oscillator is enabled.

enumerator kCCM32K_Only32kOscEnabled

Indicates only 32kHz crystal oscillator is enabled.

enumerator kCCM32K_Both32kFro32kOscEnabled

Indicates both 32kHz free running oscillator and 32kHz crystal oscillator are enabled.

enum _ccm32k_clock_source

The enumerator of clock source.

Values:

enumerator kCCM32K_ClockSourceNone

None clock source.

enumerator kCCM32K_ClockSource32kFro

32kHz free running oscillator is the clock source.

enumerator kCCM32K_ClockSource32kOsc

32kHz crystal oscillator is the clock source.

enum _ccm32k_clock_monitor_freq_trim_value

Clock monitor frequency trim values.

Values:

enumerator kCCM32K_ClockMonitor2CycleAssert

Clock monitor asserts 2 cycle after expected edge (assert after 10 cycles with no edge).

enumerator kCCM32K_ClockMonitor4CycleAssert

Clock monitor asserts 4 cycle after expected edge (assert after 12 cycles with no edge).

enumerator kCCM32K_ClockMonitor6CycleAssert

Clock monitor asserts 6 cycle after expected edge (assert after 14 cycles with no edge).

enumerator kCCM32K_ClockMonitor8CycleAssert

Clock monitor asserts 8 cycle after expected edge (assert after 16 cycles with no edge).

enum _ccm32k_clock_monitor_divide_trim_value

Clock monitor divide trim values.

Values:

enumerator kCCM32K_ClockMonitor_1kHzFro32k_1kHzOsc32k

Clock monitor operates at 1 kHz for both FRO32K and OSC32K.

enumerator kCCM32K_ClockMonitor_64HzFro32k_1kHzOsc32k

Clock monitor operates at 64 Hz for FRO32K and clock monitor operates at 1 kHz for OSC32K.

enumerator kCCM32K_ClockMonitor_1KHzFro32k_64HzOsc32k

Clock monitor operates at 1K Hz for FRO32K and clock monitor operates at 64 Hz for OSC32K.

enumerator kCCM32K_ClockMonitor_64HzFro32k_64HzOsc32k

Clock monitor operates at 64 Hz for FRO32K and clock monitor operates at 64 Hz for OSC32K.

enum _ccm32k_clock_source_select

CCM32K clock source enumeration.

Values:

enumerator kCCM32K_ClockSourceSelectFro32k

FRO32K clock output is selected as clock source.

enumerator kCCM32K_ClockSourceSelectOsc32k

OSC32K clock output is selected as clock source.

enum _ccm32k_clock_output_peripheral

32kHz clock output peripheral bit map.

Values:

enumerator kCCM32K_ClockOutToRtc

32kHz clock output to RTC.

enumerator kCCM32K_ClockOutToRfmc

32kHz clock output to Rfmc.

enumerator kCCM32K_ClockOutToNbu

32kHz clock output to NBU.

enumerator kCCM32K_ClockOutToWuuRmcPortD

32kHz clock output to WUU/RMC/PORTD.

enumerator kCCM32K_ClockOutToOtherModules

32kHz clock output to Other modules.

typedef enum _ccm32k_osc_xtal_cap ccm32k_osc_xtal_cap_t

The enumerator of internal capacitance of OSC's XTAL pin.

typedef enum _ccm32k_osc_extal_cap ccm32k_osc_extal_cap_t

The enumerator of internal capacitance of OSC's EXTAL pin.

typedef enum _ccm32k_osc_fine_adjustment_value ccm32k_osc_fine_adjustment_value_t

The enumerator of osc amplifier gain fine adjustment. Changes the oscillator amplitude by modifying the automatic gain control (AGC).

typedef enum _ccm32k_osc_coarse_adjustment_value ccm32k_osc_coarse_adjustment_value_t

The enumerator of osc amplifier coarse fine adjustment. Tunes the internal transconductance (gm) by increasing the current.

typedef enum _ccm32k_osc_mode ccm32k_osc_mode_t

The enumerator of 32kHz oscillator.

typedef enum _ccm32k_state ccm32k_state_t

The enumerator of module state.

typedef enum _ccm32k_clock_source ccm32k_clock_source_t

The enumerator of clock source.

typedef enum _ccm32k_clock_monitor_freq_trim_value

ccm32k_clock_monitor_freq_trim_value_t

Clock monitor frequency trim values.

```
typedef enum _ccm32k_clock_monitor_divide_trim_value
```

```
ccm32k_clock_monitor_divide_trim_value_t
```

Clock monitor divide trim values.

```
typedef struct _ccm32k_clock_monitor_config ccm32k_clock_monitor_config_t
```

Clock monitor configuration structure.

```
typedef enum _ccm32k_clock_source_select ccm32k_clock_source_select_t
```

CCM32K clock source enumeration.

```
typedef enum _ccm32k_clock_output_peripheral ccm32k_clock_output_peripheral_t
```

32kHz clock output peripheral bit map.

```
typedef struct _ccm32k_osc_config ccm32k_osc_config_t
```

The structure of oscillator configuration.

```
CCM32K_OSC32K_CTRL_OSC_MODE_MASK
```

```
CCM32K_OSC32K_CTRL_OSC_MODE_SHIFT
```

```
CCM32K_OSC32K_CTRL_OSC_MODE(x)
```

```
struct _ccm32k_clock_monitor_config
```

#include <fsl_ccm32k.h> Clock monitor configuration structure.

Public Members

```
bool enableClockMonitor
```

Used to enable/disable clock monitor.

```
ccm32k_clock_monitor_freq_trim_value_t freqTrimValue
```

Clock minitor frequency trim value.

```
ccm32k_clock_monitor_divide_trim_value_t divideTrimValue
```

Clock minitor divide trim value.

```
struct _ccm32k_osc_config
```

#include <fsl_ccm32k.h> The structure of oscillator configuration.

Public Members

```
bool enableInternalCapBank
```

enable/disable the internal capacitance bank.

```
ccm32k_osc_xtal_cap_t xtalCap
```

The internal capacitance for the OSC XTAL pin from the capacitor bank, only useful when the internal capacitance bank is enabled.

```
ccm32k_osc_extal_cap_t extalCap
```

The internal capacitance for the OSC EXTAL pin from the capacitor bank, only useful when the internal capacitance bank is enabled.

```
ccm32k_osc_fine_adjustment_value_t fineAdjustment
```

32kHz crystal oscillator amplifier fine adjustment value.

```
ccm32k_osc_coarse_adjustment_value_t coarseAdjustment
```

32kHz crystal oscillator amplifier coarse adjustment value.

2.3 CE Basic Functions

`int32_t CE_ExecCmd(void)`

Executes commands in the command queue.

Returns

Command execution status.

`int32_t CE_NullCmd(void)`

Sends a basic command to verify that ZV2117 is operational and that the command interface is functioning properly.

This function passes through the command processing interface and returns a success status.

Returns

Command execution status.

`int32_t CE_Copy(int32_t *pDst, int32_t *pSrc, const int32_t N)`

Copies data between buffers in the ZV2117 memory section.

Copies a specified number of 32-bit words from the source to the destination buffer. Both buffers must reside in the ZV2117 data memory section.

Parameters

- `pDst` – **[out]** Pointer to destination buffer
- `pSrc` – **[in]** Pointer to source buffer
- `N` – **[in]** Number of 32-bit words to copy

Returns

Command execution status.

2.4 CE Command Functions

`int32_t CE_CmdInitBuffer(ce_cmdbuffer_t *psCmdBuffer, volatile uint32_t cmdbuffer[], volatile int32_t statusbuffer[], ce_cmd_mode_t cmdmode)`

Initializes the CM33-ZV2117 command buffer.

This function must be called once after power-up or reset, or when the command queue mode needs to be changed. Once configured, the command mode remains unchanged till reset or re-initialization.

Parameters

- `psCmdBuffer` – **[out]** Pointer to the command buffer structure. Must be allocated in ARM memory (not ZV2117).
- `cmdbuffer` – **[in]** Command buffer. Must be 256 words in ZV2117 data memory.
- `statusbuffer` – **[in]** Status buffer. Must be 134 words in ZV2117 data memory.
- `cmdmode` – **[in]** Command mode. One of:
 - `kCE_CmdModeOneNonBlocking`
 - `kCE_CmdModeMultipleNonBlocking`
 - `kCE_CmdModeOneBlocking`
 - `kCE_CmdModeMultipleBlocking`

Return values

0 – Initialization is successful.

`int32_t CE_CmdReset(void)`

Resets the CM33-ZV2117 command queue.

Any pending commands in the queue will be flushed.

Return values

0 – Reset is successful.

`int32_t CE_CmdAdd(ce_cmd_t cmd, ce_cmdstruct_t *cmdargs)`

Adds a command to the command queue.

Parameters

- `cmd` – **[in]** Command name. Choose from the enum description in `fsl_ce_if.h`. Not all of the cmd are implemented in the current release.
- `cmdargs` – **[in]** Arguments structure detailing the arguments for the function/command.

Return values

- 0 – Command added successfully.
- -1 – Command not added (queue is full).

`int32_t CE_CmdLaunch(int32_t force_launch)`

Launches the ZV2117 with the current command queue.

Parameters

- `force_launch` – **[in]**
 - 1: Launches the queue regardless of the command mode.
 - 0: Launches only if in single-command mode. Otherwise, does nothing.

Return values

0 – Launch is successful.

`int32_t CE_CmdLaunchBlocking(void)`

Launches the current command queue and waits for completion.

Return values

0 – Launch is successful.

`int32_t CE_CmdLaunchNonBlocking(void)`

Launches the current command queue and returns immediately.

ZV2117 will send an interrupt via MUA->GCR to ARM upon task completion. Alternatively, the user can poll for completion.

If using interrupt, the user must call `CE_CmdReset()` in the IRQ handler. `IRQ::DSP_IRQn` must be enabled.

The user can optionally also poll to figure out the command queue execution status.

Return values

0 – Launch is successful.

`int32_t CE_CmdCheckStatus(void)`

Checks the execution status of the current command queue. Only applicable in non-blocking mode.

Returns

- `CE_STATUS_BUSY` ZV2117 is still executing.

- CE_STATUS_IDLE Execution completed; ZV2117 is ready for new commands.

CE__COMPUTE_TIMEOUT

Maximum loop wait time for CE computation.

When CE is computing, driver will wait for the computation to complete. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

2.5 CE CMSIS Functions

```
void ce_arm_cfft_f32(const arm_cfft_instance_f32 *S, float *p1, uint8_t ifftFlag, uint8_t  
                    bitReverseFlag, float *pOut, float *pScratch)
```

CMSIS-compatible FFT implementation using ZV2117 backend.

This API is designed to be compatible with the ARM CM33 FFT interface. It delegates the FFT computation to the ZV2117 CE module. Please refer to CM33 documentation for details.

Note: This API only support float32 FFTs.

```
struct arm_cfft_instance_f32
```

```
#include <fsl_ce_cmsis.h> FFT/IFFT float32.
```

Public Members

```
uint16_t fftLen
```

Length of the FFT.

```
const float *pTwiddle
```

Points to the twiddle factor table.

```
const uint16_t *pBitRevTable
```

Points to the bit reversal table.

```
uint16_t bitRevLength
```

Length of the bit reversal table.

2.6 CE Initialization Functions

```
FSL_CE_DRIVER_VERSION
```

CE driver version.

```
typedef struct _ce_copy_image ce_copy_image_t
```

Structure for CE copy image to destination address.

Defines source and destination address for copying image with given size.

```
void CE_Init(ce_copy_image_t *ceCopyImage)
```

Initializes the CE.

Parameters

- ceCopyImage – **[in]** The information about the CE image to copy.

void CE_InstallFirmware(*ce_copy_image_t* *ceCopyImage)

Installs CE firmware by given image info.

Parameters

- ceCopyImage – **[in]** The information about the CE image to copy.

void CE_InitWithoutFirmware(void)

Initializes the CE.

This function is similar to CE_Init, but it does not install the firmware, the firmware can be installed using CE_InstallFirmware.

struct __ce_copy_image

#include <fsl_ce.h> Structure for CE copy image to destination address.

Defines source and destination address for copying image with given size.

2.7 CE Matrix Functions

int32_t CE_MatrixAdd_Q15(int16_t *pDst, int16_t *pA, int16_t *pB, int32_t M, int32_t N)

Calculates the sum of two real 16-bit integer (Q15) matrices.

Computes $C = A + B$, where A, B, and C are an $M \times N$ real int16_t matrices. All matrices are assumed to have the same format and dimensions.

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times N$).
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $M \times N$).
- M – **[in]** Number of rows of matrices A, B, and C.
- N – **[in]** Number of columns of matrices A, B, and C.

Returns

Command execution status.

int32_t CE_MatrixAdd_Q31(int32_t *pDst, int32_t *pA, int32_t *pB, int32_t M, int32_t N)

Calculates the sum of two real 32-bit integer (Q31) matrices.

Computes $C = A + B$, where A, B, and C are an $M \times N$ real int32_t matrices. All matrices are assumed to have the same format and dimensions.

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times N$).
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $M \times N$).
- M – **[in]** Number of rows of matrices A, B, and C.
- N – **[in]** Number of columns of matrices A, B, and C.

Returns

Command execution status.

int32_t CE_MatrixAdd_F32(float *pDst, float *pA, float *pB, int32_t M, int32_t N)

Calculates the sum of two real 32-bit floating-point matrices.

Computes $C = A + B$, where A, B, and C are an $M \times N$ real float32 matrices. All matrices are assumed to have the same format and dimensions.

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times N$).
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $M \times N$).
- M – **[in]** Number of rows of matrices A, B, and C.
- N – **[in]** Number of columns of matrices A, B, and C.

Returns

Command execution status.

int32_t CE_MatrixElemMul_F32(float *pDst, float *pA, float *pB, int32_t M, int32_t N)

Calculates the element-wise product of two real 32-bit floating-point matrices.

Computes $C = A \times B$, where A, B, and C are an $M \times N$ real float32 matrices. All matrices are assumed to have the same format and dimensions.

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times N$).
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $M \times N$).
- M – **[in]** Number of rows of matrices A, B, and C.
- N – **[in]** Number of columns of matrices A, B, and C.

Returns

Command execution status.

int32_t CE_MatrixMul_F32(float *pDst, float *pA, float *pB, int32_t M, int32_t N, int32_t P)

Calculates the matrix product of two real 32-bit floating-point matrices.

Computes $C = A \times B$, where:

- A is an $M \times N$ real float32 matrix,
- B is an $N \times P$ real float32 matrix,
- C is the resulting $M \times P$ real float32 matrix.

All matrices are assumed to be in row-major format.

Note: Data precision and format is as defined by the argument type. Limits on max value of N:

- For F32: $N < 128$;
 - For CF32: $N < 64$.
-

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times P$)
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $N \times P$).
- M – **[in]** Number of rows of matrix A.
- N – **[in]** Number of columns in matrix A (and rows in matrix B).
- P – **[in]** Number of columns of matrix B.

Returns

Command execution status.

`int32_t CE_MatrixMul_CF32(float *pDst, float *pA, float *pB, int32_t M, int32_t N, int32_t P)`

Calculates the matrix product of two complex 32-bit floating point matrices.

Computes $C = A \times B$, where:

- A is an $M \times N$ complex float32 matrix,
- B is an $N \times P$ complex float32 matrix,
- C is the resulting $M \times P$ complex float32 matrix.

All matrices are assumed to be in row-major format.

Parameters

- pDst – **[out]** Pointer to the output matrix C (size $M \times P$).
- pA – **[in]** Pointer to the input matrix A (size $M \times N$).
- pB – **[in]** Pointer to the input matrix B (size $N \times P$).
- M – **[in]** Number of rows of matrix A.
- N – **[in]** Number of columns in matrix A (and rows in matrix B).
- P – **[in]** Number of columns of matrix B.

Returns

Command execution status.

`int32_t CE_MatrixInvHerm_CF32(float *pAinv, float *pA, float *pScratch, int32_t M, uint8_t flag_packedInput, uint8_t flag_cholInv)`

Calculates the inverse or Cholesky inverse of a complex Hermitian matrix (in float32 precision).

Computes one of the following:

- $A_{inv} = \text{inv}(A)$
- $A_{inv} = \text{inv}(\text{chol}(A))$

where $\text{chol}(A)$ is the lower triangular Cholesky decomposition of matrix A.

The input matrix must be in row-major format and can be either:

- A full $M \times M$ matrix, or
- A packed upper triangular matrix (containing only the upper triangle).

The output matrix is written in row-major format, and only the lower triangle is stored. The total number of output elements is:

- $M_c = (M + 1) \times M / 2$

Note: The input, output and scratch buffers must be separately allocated and non-overlapping.

Parameters

- pAinv – **[out]** Pointer to the output matrix inverse. M_c elements are written (complex float32).
- pA – **[in]** Pointer to the input matrix A (complex float32, size $M \times M$).
- pScratch – **[in]** Pointer to scratch buffer (minimum size $M_c \times 3 \times 8$ bytes).
- M – **[in]** Number of rows/columns in matrix A.

- `flag_packedInput` – **[in]** Format of input matrix:
 - 0: full matrix
 - 1: packed upper triangular matrix
- `flag_cholInv` – **[in]** Type of inverse to compute:
 - 0: $\text{inv}(A)$
 - 1: $\text{inv}(\text{chol}(A))$

Returns

Command execution status.

`int32_t CE_MatrixEvdHerm_CF32(float *pLambdaOut, float *pUout, float *pUin, float *pScratch, int32_t M, float tol, int32_t max_iter, uint8_t flag_packedInput)`

Calculates the Eigenvalue Decomposition (EVD) of a complex Hermitian matrix (float32 precision).

Performs EVD on an $M \times M$ complex Hermitian matrix A:

- $[U, T] = \text{eig}(A)$

The input matrix must be in column-major format and can be either:

- A full $M \times M$ matrix, or
- A packed upper triangular matrix.

The decomposition uses an iterative QR algorithm to compute:

- U: matrix of eigenvectors ($M \times M$, column-major)
- T: vector of eigenvalues ($M \times 1$)

Maximum supported matrix size: $M \leq 40$.

Note: If input and output buffers overlap, the output U matrix will overwrite the input matrix A.

Parameters

- `pLambdaOut` – **[out]** Pointer to the output vector of eigenvalues (size $M \times 1$, complex float32).
- `pUout` – **[out]** Pointer to the output matrix of eigenvectors (size $M \times M$, column-major, complex float32).
- `pUin` – **[in]** Pointer to the input matrix A (size $M \times M$, column-major, complex float32).
- `pScratch` – **[in]** Pointer to scratch buffer (minimum size: $(M \times M \times 4 + 360) \times 4$ bytes).
- `M` – **[in]** Number of rows/columns in matrix A.
- `tol` – **[in]** Tolerance value for QR convergence. Smaller values yield better accuracy but require more iterations.
- `max_iter` – **[in]** Maximum number of QR iterations. If exceeded, the function terminates regardless of convergence.
- `flag_packedInput` – **[in]** Format of input matrix:
 - 0: full matrix
 - 1: packed upper triangular part only

Returns

Command execution status. The number of QR iterations is returned in status[3] register.

```
int32_t CE_MatrixChol_CF32(float *pL, float *pA, float *pScratch, int32_t M, uint8_t
    flag_packedInput)
```

Calculates the Cholesky Decomposition of a complex Hermitian matrix (float32 precision).

Calculates $L = \text{chol}(A)$, where A is a complex Hermitian $M \times M$ matrix and L is a lower triangular matrix such that $A = L \times L^H$.

The input matrix must be in row-major format and can be either:

- A full $M \times M$ matrix, or
- A packed format containing only the upper triangular elements.

The output matrix is also in row-major format, and only the lower triangular part is written. The total number of output elements is M_c , where:

- $M_c = (M + 1) \times M / 2$

Note: The input, output and scratch buffers must be separately allocated and non-overlapping.

Parameters

- pL – **[out]** Pointer to the output lower triangular matrix L . Only M_c elements are written (complex-valued float32 data in row-major format).
- pA – **[in]** Pointer to the input matrix A (full or packed format). Must be complex-valued float32 data in row-major format.
- $pScratch$ – **[in]** Pointer to a scratch buffer (at least $M_c \times 8$ bytes).
- M – **[in]** Number of rows/columns of matrix A .
- flag_packedInput – **[in]** Format flag for input matrix:
 - 0: full matrix
 - 1: packed upper triangular matrix

Returns

Command execution status.

2.8 CE Transform Functions

```
int32_t CE_TransformCFFT_F16(float *pY, float *pX, float *pScratch, int32_t log2N)
```

Computes the FFT of complex-valued 16-bit floating point data.

Performs an N -point Fast Fourier Transform (FFT) on a complex-valued float16 input stream. The FFT size N must be a power of 2, with valid range: $16 \leq N \leq 16384$.

Note:

- Data precision and format follow the argument types.
 - `float*` is used to represent float16 pointers.
 - The input, output and scratch buffers must be separately allocated and non-overlapping.
-

Parameters

- pY – **[out]** Pointer to the FFT output buffer (complex-valued float16 data).
- pX – **[in]** Pointer to the FFT input buffer (complex-valued float16 data).
- pScratch – **[in]** Pointer to a scratch buffer (at least $N \times 4$ bytes).
- log2N – **[in]** Base-2 logarithm of the FFT size N.

Returns

Command execution status.

int32_t CE_TransformCFFT_F32(float *pY, float *pX, float *pScratch, int32_t log2N)

Computes the FFT of complex-valued 32-bit floating point data.

Performs an N-point Fast Fourier Transform (FFT) on a complex-valued float32 input stream. The FFT size N must be a power of 2, with valid range: $32 \leq N \leq 16384$.

Note: The input, output and scratch buffers must be separately allocated and non-overlapping.

Parameters

- pY – **[out]** Pointer to the FFT output buffer (complex-valued float32).
- pX – **[in]** Pointer to the FFT input buffer (complex-valued float32).
- pScratch – **[in]** Pointer to a scratch buffer (at least $N \times 8$ bytes).
- log2N – **[in]** Base-2 logarithm of the FFT size N.

Returns

Command execution status.

int32_t CE_TransformIFFT_F16(float *pY, float *pX, float *pScratch, int32_t log2N)

Computes the IFFT of complex-valued 16-bit floating point data.

Performs an N-point Inverse Fast Fourier Transform (IFFT) on a complex-valued float16 input stream. The IFFT size N must be a power of 2, with valid range: $16 \leq N \leq 16384$.

Note:

- Data precision and format follow the argument types.
 - float* is used to represent float16 pointers.
 - The input, output and scratch buffers must be separately allocated and non-overlapping.
-

Parameters

- pY – **[out]** Pointer to the IFFT output buffer (complex-valued float16).
- pX – **[in]** Pointer to the IFFT input buffer (complex-valued float16).
- pScratch – **[in]** Pointer to a scratch buffer (at least $N \times 4$ bytes).
- log2N – **[in]** Base-2 logarithm of the IFFT size N.

Returns

Command execution status.

int32_t CE_TransformIFFT_F32(float *pY, float *pX, float *pScratch, int32_t log2N)

Computes the IFFT of complex-valued 32-bit floating point data.

Performs an N-point Inverse Fast Fourier Transform (IFFT) on a complex-valued float32 input stream. The IFFT size N must be a power of 2, with valid range: $32 \leq N \leq 16384$.

Note: The input, output and scratch buffers must be separately allocated and non-overlapping.

Parameters

- pY – **[out]** Pointer to the IFFT output buffer (complex-valued float32).
- pX – **[in]** Pointer to the IFFT input buffer (complex-valued float32).
- pScratch – **[in]** Pointer to a scratch buffer (at least $N \times 8$ bytes).
- log2N – **[in]** Base-2 logarithm of the IFFT size N.

Returns

Command execution status.

2.9 Clock Driver

uint32_t CLOCK_GetSysClkFreq(*scg_sys_clk_t* type)

Gets the SCG system clock frequency.

This function gets the SCG system clock frequency. These clocks are used for core, platform, external, and bus clock domains.

Parameters

- type – Which type of clock to get, core clock or slow clock.

Returns

Clock frequency.

static inline void CLOCK_SetRunModeSysClkConfig(const *scg_sys_clk_config_t* *config)

Sets the system clock configuration for RUN mode.

This function sets the system clock configuration for RUN mode.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetCurSysClkConfig(*scg_sys_clk_config_t* *config)

Gets the system clock configuration in the current power mode.

This function gets the system configuration in the current power mode.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_SetClkOutSel(*clock_clkout_src_t* setting)

Sets the clock out selection.

This function sets the clock out selection (CLKOUTSEL).

Parameters

- setting – The selection to set.

status_t CLOCK_InitSysOsc(const *scg_sosc_config_t* *config)

Initializes the SCG system OSC.

This function enables the SCG system OSC clock according to the configuration.

Note: This function can't detect whether the system OSC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – System OSC is initialized.
- kStatus_SCG_Busy – System OSC has been enabled and is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

status_t CLOCK_DeinitSysOsc(void)

De-initializes the SCG system OSC.

This function disables the SCG system OSC clock.

Note: This function can't detect whether the system OSC is used by an IP.

Return values

- kStatus_Success – System OSC is deinitialized.
- kStatus_SCG_Busy – System OSC is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

uint32_t CLOCK_GetSysOscFreq(void)

Gets the SCG system OSC clock frequency (SYSOSC).

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsSysOscErr(void)

Checks whether the system OSC clock error occurs.

Returns

True if the error occurs, false if not.

static inline void CLOCK_ClearSysOscErr(void)

Clears the system OSC clock error.

static inline void CLOCK_SetSysOscMonitorMode(*scg_sosc_monitor_mode_t* mode)

Sets the system OSC monitor mode.

This function sets the system OSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

static inline bool CLOCK_IsSysOscValid(void)

Checks whether the system OSC clock is valid.

Returns

True if clock is valid, false if not.

static inline void CLOCK_UnlockSysOscControlStatusReg(void)

Unlock the SOSCCSR control status register.

static inline void CLOCK_LockSysOscControlStatusReg(void)

Lock the SOSCCSR control status register.

status_t CLOCK_InitSirc(const scg_sirc_config_t *config)

Initializes the SCG slow IRC clock.

This function enables the SCG slow IRC clock according to the configuration.

Note: This function can't detect whether the system OSC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – SIRC is initialized.
- kStatus_SCG_Busy – SIRC has been enabled and is used by system clock.
- kStatus_ReadOnly – SIRC control register is locked.

status_t CLOCK_DeinitSirc(void)

De-initializes the SCG slow IRC.

This function disables the SCG slow IRC.

Note: This function can't detect whether the SIRC is used by an IP.

Return values

- kStatus_Success – SIRC is deinitialized.
- kStatus_SCG_Busy – SIRC is used by system clock.
- kStatus_ReadOnly – SIRC control register is locked.

uint32_t CLOCK_GetSircFreq(void)

Gets the SCG SIRC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsSircValid(void)

Checks whether the SIRC clock is valid.

Returns

True if clock is valid, false if not.

static inline void CLOCK_UnlockSircControlStatusReg(void)

Unlock the SIRCCSR control status register.

`static inline void CLOCK_LockSircControlStatusReg(void)`

Lock the SIRCCSR control status register.

`status_t CLOCK_InitFirc(const scg_firc_config_t *config)`

Initializes the SCG fast IRC clock.

This function enables the SCG fast IRC clock according to the configuration.

Note: This function can't detect whether the FIRC has been enabled and used by an IP.

Parameters

- `config` – Pointer to the configuration structure.

Return values

- `kStatus_Success` – FIRC is initialized.
- `kStatus_SCG_Busy` – FIRC has been enabled and is used by the system clock.
- `kStatus_ReadOnly` – FIRC control register is locked.

`status_t CLOCK_DeinitFirc(void)`

De-initializes the SCG fast IRC.

This function disables the SCG fast IRC.

Note: This function can't detect whether the FIRC is used by an IP.

Return values

- `kStatus_Success` – FIRC is deinitialized.
- `kStatus_SCG_Busy` – FIRC is used by the system clock.
- `kStatus_ReadOnly` – FIRC control register is locked.

`uint32_t CLOCK_GetFircFreq(void)`

Gets the SCG FIRC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`static inline bool CLOCK_IsFircErr(void)`

Checks whether the FIRC clock error occurs.

Returns

True if the error occurs, false if not.

`static inline void CLOCK_ClearFircErr(void)`

Clears the FIRC clock error.

`static inline bool CLOCK_IsFircValid(void)`

Checks whether the FIRC clock is valid.

Returns

True if clock is valid, false if not.

`static inline void CLOCK_UnlockFircControlStatusReg(void)`

Unlock the FIRCCSR control status register.

`static inline void CLOCK__LockFireControlStatusReg(void)`

Lock the FIRCCSR control status register.

`static inline bool CLOCK__IsFIRCAutoTrimLocked(void)`

Check whether FIRC auto trim locked to target frequency range.

When FIRCTREN and FIRCTRUP are enabled, TRIM_LOCK will indicate when auto trimming is complete and output FIRC frequency has locked to target FIRC range. TRIM_LOCK will automatically get cleared if FIRCTREN and FIRCTRUP are not set.

Returns

True if FIRC trim locked to target frequency range, false if not.

`status_t CLOCK__InitRosc(const scg_rosc_config_t *config)`

brief Initializes the SCG ROSC.

This function enables the SCG ROSC clock according to the configuration.

param config Pointer to the configuration structure. retval kStatus_Success ROSC is initialized. retval kStatus_SCG_Busy ROSC has been enabled and is used by the system clock. retval kStatus_ReadOnly ROSC control register is locked.

note This function can't detect whether the system OSC has been enabled and used by an IP.

`status_t CLOCK__DeinitRosc(void)`

brief De-initializes the SCG ROSC.

This function disables the SCG ROSC clock.

retval kStatus_Success System OSC is deinitialized. retval kStatus_SCG_Busy System OSC is used by the system clock. retval kStatus_ReadOnly System OSC control register is locked.

note This function can't detect whether the ROSC is used by an IP.

`uint32_t CLOCK__GetRtcOscFreq(void)`

Gets the SCG RTC OSC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`status_t CLOCK__InitRfFro192M(const fro192m_rf_clk_config_t *config)`

Initializes the FRO192M clock for the Radio Mode Controller.

This function configure the RF FRO192M clock according to the configuration.

Parameters

- config – Pointer to the configuration structure.

Return values

kStatus_Success – RF FRO192M is configured.

`uint32_t CLOCK__GetRfFro192MFreq(void)`

Gets the FRO192M clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`static inline bool CLOCK__IsRoscErr(void)`

Checks whether the ROSC clock error occurs.

Returns

True if the error occurs, false if not.

`static inline void CLOCK__ClearRoscErr(void)`

Clears the ROSC clock error.

```
static inline void CLOCK_SetRoscMonitorMode(scg_rosc_monitor_mode_t mode)
```

Sets the ROSC monitor mode.

This function sets the ROSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

```
static inline bool CLOCK_IsRoscValid(void)
```

Checks whether the ROSC clock is valid.

Returns

True if clock is valid, false if not.

```
static inline void CLOCK_UnlockRoscControlStatusReg(void)
```

Unlock the ROSCCSR control status register.

```
static inline void CLOCK_LockRoscControlStatusReg(void)
```

Lock the ROSCCSR control status register.

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

Sets the XTAL0 frequency based on board settings.

Parameters

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

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

Sets the XTAL32 frequency based on board settings.

Parameters

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

```
FSL_CLOCK_DRIVER_VERSION
```

CLOCK driver version 2.2.6.

```
enum _clock_name
```

Clock name used to get clock frequency.

These clocks source would be generated from SCG module.

Values:

```
enumerator kCLOCK_CoreSysClk
```

Cortex M33 clock.

```
enumerator kCLOCK_SlowClk
```

SLOW_CLK with DIVSLOW.

```
enumerator kCLOCK_PlatClk
```

PLAT_CLK.

```
enumerator kCLOCK_SysClk
```

SYS_CLK.

```
enumerator kCLOCK_BusClk
```

BUS_CLK with DIVBUS.

```
enumerator kCLOCK_ScgSysOscClk
```

SCG system OSC clock.

```
enumerator kCLOCK_ScgSircClk
```

SCG SIRC clock.

enumerator kCLOCK_ScgFircClk
SCG FIRC clock.

enumerator kCLOCK_RtcOscClk
RTC OSC clock.

enum _clock_ip_control

Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[CC]

Values:

enumerator kCLOCK_IpClkControl_fun0
Peripheral clocks are disabled, module does not stall low power mode entry.

enumerator kCLOCK_IpClkControl_fun1
Peripheral clocks are enabled, module does not stall low power mode entry.

enumerator kCLOCK_IpClkControl_fun2
Peripherals clocks are enabled unless peripheral is idle, low power mode entry will stall until peripheral is idle.

enumerator kCLOCK_IpClkControl_fun3
Peripheral clocks are enabled unless in SLEEP mode (or lower), low power mode entry will stall until peripheral is idle Peripheral functional clocks that remain enabled in SLEEP mode are enabled and do not stall low power mode entry unless entering DEEPSLEEP mode (or lower)

enum _clock_ip_src

Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[MUX].

Values:

enumerator kCLOCK_IpSrcFro6M
FRO 6M clock.

enumerator kCLOCK_IpSrcFro192M
FRO 192M clock.

enumerator kCLOCK_IpSrcSoscClk
OSC RF clock.

enumerator kCLOCK_IpSrc32kClk
32k Clk clock.

enum _tpm2_ip_src

Clock source for TPM2.

These options are for RF_CMC1->TPM2_CFG[CLK_MUX_SEL].

Values:

enumerator kCLOCK_Tpm2SrcCoreClk
Core Clock.

enumerator kCLOCK_Tpm2SrcSoscClk
Radio Oscillator.

enum _clock_ip_name

Clock IP name.

Values:

enumerator kCLOCK_NOGATE
No clock gate for the IP in MRCC

enumerator kCLOCK_Ewm0
Clock ewm0

enumerator kCLOCK_Syspm0
Clock syspm0

enumerator kCLOCK_Wdog0
Clock wdog0

enumerator kCLOCK_Wdog1
Clock wdog1

enumerator kCLOCK_Sfa0
Clock sfa0

enumerator kCLOCK_Crc0
Clock crc0

enumerator kCLOCK_Secsubsys
Clock secsubsys

enumerator kCLOCK_Lpit0
Clock lpit0

enumerator kCLOCK_Tstmr0
Clock tstmr0

enumerator kCLOCK_Tpm0
Clock tpm0

enumerator kCLOCK_Tpm1
Clock tpm1

enumerator kCLOCK_Lpi2c0
Clock lpi2c0

enumerator kCLOCK_Lpi2c1
Clock lpi2c1

enumerator kCLOCK_I3c0
Clock i3c

enumerator kCLOCK_Lpspi0
Clock lpspi0

enumerator kCLOCK_Lpspi1
Clock lpspi1

enumerator kCLOCK_Lpuart0
Clock lpuart0

enumerator kCLOCK_Lpuart1
Clock lpuart1

enumerator kCLOCK_Flexio0
Clock Flexio0

enumerator kCLOCK_Can0
Clock Can0

enumerator kCLOCK_Sema0
Clock Sema0

enumerator kCLOCK_Data_stream_2p4
Clock data_stream_2p4

enumerator kCLOCK_PortA
Clock portA

enumerator kCLOCK_PortB
Clock portB

enumerator kCLOCK_PortC
Clock portC

enumerator kCLOCK_Lpadc0
Clock lpadc0

enumerator kCLOCK_Lpcmp0
Clock lpcmp0

enumerator kCLOCK_Lpcmp1
Clock lpcmp1

enumerator kCLOCK_Vref0
Clock verf0

enumerator kCLOCK_Mtr_master
Clock mtr_master

enumerator kCLOCK_Can1
Clock Can1

enumerator kCLOCK_GpioA
Clock gpioA

enumerator kCLOCK_GpioB
Clock gpioB

enumerator kCLOCK_GpioC
Clock gpioC

enumerator kCLOCK_Dma0
Clock dma0

enumerator kCLOCK_Pflexnvm
Clock pflexnvm

enumerator kCLOCK_Sram0
Clock Sram0

enumerator kCLOCK_Sram1
Clock Sram1

enumerator kCLOCK_Sram2
Clock Sram2

enumerator kCLOCK_Sram3
Clock Sram3

enumerator kCLOCK_Sram0_NOECC
Clock Sram0 NOECC

enumerator kCLOCK_DSP0

Clock DSPV

enumerator kCLOCK_DSP0_MUA

Clock DSPV MUA

enumerator kCLOCK_Sram1_NOECC

Clock Sram1 NOECC

enumerator kCLOCK_Rf_2p4ghz_bist

Clock rf_2p4ghz_bist

enumerator kCLOCK_Lptmr0_wake2vsys

Clock LPTMR0 IPG clk erclk wake2vsys

enumerator kCLOCK_Lptmr1_wake2vsys

Clock LPTMR1 IPG clk erclk wake2vsys

enumerator kCLOCK_Lptmr2_wake2vsys

Clock LPTMR2 IPG clk erclk wake2vsys

enumerator kCLOCK_Sirc_vsys_gating

Clock SIRC vsys gating

enum _scg_status

SCG status return codes.

Values:

enumerator kStatus_SCG_Busy

Clock is busy.

enumerator kStatus_SCG_InvalidSrc

Invalid source.

enum _scg_sys_clk

SCG system clock type.

Values:

enumerator kSCG_SysClkSlow

System slow clock.

enumerator kSCG_SysClkBus

Bus clock.

enumerator kSCG_SysClkPlatform

Platform clock.

enumerator kSCG_SysClkCore

Core clock.

enum _scg_sys_clk_src

SCG system clock source.

ERR052742: FRO6M clock(kSCG_SysClkSrcSirc) is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

Values:

enumerator kSCG_SysClkSrcSysOsc
System OSC.

enumerator kSCG_SysClkSrcSirc
Slow IRC.

enumerator kSCG_SysClkSrcFirc
Fast IRC.

enumerator kSCG_SysClkSrcRosc
RTC OSC.

enum __scg_sys_clk_div
SCG system clock divider value.

Values:

enumerator kSCG_SysClkDivBy1
Divided by 1.

enumerator kSCG_SysClkDivBy2
Divided by 2.

enumerator kSCG_SysClkDivBy3
Divided by 3.

enumerator kSCG_SysClkDivBy4
Divided by 4.

enumerator kSCG_SysClkDivBy5
Divided by 5.

enumerator kSCG_SysClkDivBy6
Divided by 6.

enumerator kSCG_SysClkDivBy7
Divided by 7.

enumerator kSCG_SysClkDivBy8
Divided by 8.

enumerator kSCG_SysClkDivBy9
Divided by 9.

enumerator kSCG_SysClkDivBy10
Divided by 10.

enumerator kSCG_SysClkDivBy11
Divided by 11.

enumerator kSCG_SysClkDivBy12
Divided by 12.

enumerator kSCG_SysClkDivBy13
Divided by 13.

enumerator kSCG_SysClkDivBy14
Divided by 14.

enumerator kSCG_SysClkDivBy15
Divided by 15.

enumerator kSCG_SysClkDivBy16

Divided by 16.

enum _clock_clkout_src

SCG clock out configuration (CLKOUTSEL).

Values:

enumerator kClockClkoutSelScgSlow

SCG Slow clock.

enumerator kClockClkoutSelSosc

System OSC.

enumerator kClockClkoutSelSirc

Slow IRC.

enumerator kClockClkoutSelFirc

Fast IRC.

enumerator kClockClkoutSelScgRtcOsc

SCG RTC OSC clock.

enum _scg_sosc_monitor_mode

SCG system OSC monitor mode.

Values:

enumerator kSCG_SysOscMonitorDisable

Monitor disabled.

enumerator kSCG_SysOscMonitorInt

Interrupt when the SOSC error is detected.

enumerator kSCG_SysOscMonitorReset

Reset when the SOSC error is detected.

SOSC enable mode.

Values:

enumerator kSCG_SoscDisable

Disable SOSC clock.

enumerator kSCG_SoscEnable

Enable SOSC clock.

enumerator kSCG_SoscEnableInSleep

Enable SOSC in sleep mode.

enum _scg_rosc_monitor_mode

SCG ROsc monitor mode.

Values:

enumerator kSCG_RoscMonitorDisable

Monitor disabled.

enumerator kSCG_RoscMonitorInt

Interrupt when the RTC OSC error is detected.

enumerator kSCG_RoscMonitorReset

Reset when the RTC OSC error is detected.

enum _scg_sirc_enable_mode

SIRC enable mode.

Values:

enumerator kSCG_SircDisableInSleep

Disable SIRC clock.

enumerator kSCG_SircEnableInSleep

Enable SIRC in sleep mode.

enum _scg_firc_trim_mode

SCG fast IRC trim mode.

Values:

enumerator kSCG_FircTrimNonUpdate

FIRC trim enable but not enable trim value update. In this mode, the trim value is fixed to the initialized value which is defined by trimCoar and trimFine in configure structure scg_firc_trim_config_t.

enumerator kSCG_FircTrimUpdate

FIRC trim enable and trim value update enable. In this mode, the trim value is auto update.

enum _scg_firc_trim_src

SCG fast IRC trim source.

Values:

enumerator kSCG_FircTrimSrcSysOsc

System OSC.

enumerator kSCG_FircTrimSrcRtcOsc

RTC OSC (32.768 kHz).

FIRC enable mode.

Values:

enumerator kSCG_FircDisable

Disable FIRC clock.

enumerator kSCG_FircEnable

Enable FIRC clock.

enumerator kSCG_FircEnableInSleep

Enable FIRC in sleep mode.

enum _scg_firc_range

SCG fast IRC clock frequency range.

Values:

enumerator kSCG_FircRange48M

Fast IRC is trimmed to 48 MHz.

enumerator kSCG_FircRange64M

Fast IRC is trimmed to 64 MHz.

enumerator kSCG_FircRange96M

Fast IRC is trimmed to 96 MHz.

enumerator kSCG_FircRange192M
Fast IRC is trimmed to 192 MHz.

enum _fro192m_rf_range
FRO192M RF clock frequency range.

Values:

enumerator kFro192M_Range16M
FRO192M output frequenc 16 MHz.

enumerator kFro192M_Range24M
FRO192M output frequenc 24 MHz.

enumerator kFro192M_Range32M
FRO192M output frequenc 32 MHz.

enumerator kFro192M_Range48M
FRO192M output frequenc 48 MHz.

enumerator kFro192M_Range64M
FRO192M output frequenc 64 MHz.

enum _fro192m_rf_clk_div
RF Flash APB and RF_CMC clock divide.

Values:

enumerator kFro192M_ClkDivBy1
Divided by 1.

enumerator kFro192M_ClkDivBy2
Divided by 2.

enumerator kFro192M_ClkDivBy4
Divided by 4.

enumerator kFro192M_ClkDivBy8
Divided by 8.

typedef enum _clock_name clock_name_t
Clock name used to get clock frequency.
These clocks source would be generated from SCG module.

typedef enum _clock_ip_control clock_ip_control_t
Clock source for peripherals that support various clock selections.
These options are for MRCC->XX[CC]

typedef enum _clock_ip_src clock_ip_src_t
Clock source for peripherals that support various clock selections.
These options are for MRCC->XX[MUX].

typedef enum _tpm2_ip_src tpm2_src_t
Clock source for TPM2.
These options are for RF_CMC1->TPM2_CFG[CLK_MUX_SEL].

typedef enum _clock_ip_name clock_ip_name_t
Clock IP name.

typedef enum _scg_sys_clk scg_sys_clk_t
SCG system clock type.

`typedef enum _scg_sys_clk_src scg_sys_clk_src_t`
SCG system clock source.

ERR052742: FRO6M clock(kSCG_SysClkSrcSirc) is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

`typedef enum _scg_sys_clk_div scg_sys_clk_div_t`
SCG system clock divider value.

`typedef struct _scg_sys_clk_config scg_sys_clk_config_t`
SCG system clock configuration.

`typedef enum _clock_clkout_src clock_clkout_src_t`
SCG clock out configuration (CLKOUTSEL).

`typedef enum _scg_sosc_monitor_mode scg_sosc_monitor_mode_t`
SCG system OSC monitor mode.

`typedef struct _scg_sosc_config scg_sosc_config_t`
SCG system OSC configuration.

`typedef enum _scg_rosc_monitor_mode scg_rosc_monitor_mode_t`
SCG ROSC monitor mode.

`typedef struct _scg_rosc_config scg_rosc_config_t`
SCG ROSC configuration.

`typedef enum _scg_sirc_enable_mode scg_sirc_enable_mode_t`
SIRC enable mode.

`typedef struct _scg_sirc_config scg_sirc_config_t`
SCG slow IRC clock configuration.

`typedef enum _scg_firc_trim_mode scg_firc_trim_mode_t`
SCG fast IRC trim mode.

`typedef enum _scg_firc_trim_src scg_firc_trim_src_t`
SCG fast IRC trim source.

`typedef struct _scg_firc_trim_config scg_firc_trim_config_t`
SCG fast IRC clock trim configuration.

`typedef enum _scg_firc_range scg_firc_range_t`
SCG fast IRC clock frequency range.

`typedef struct _scg_firc_config_t scg_firc_config_t`
SCG fast IRC clock configuration.

`typedef enum _fro192m_rf_range fro192m_rf_range_t`
FRO192M RF clock frequency range.

`typedef enum _fro192m_rf_clk_div fro192m_rf_clk_div_t`
RF Flash APB and RF_CMC clock divide.

`typedef struct _fro192m_rf_clk_config fro192m_rf_clk_config_t`
FRO192M RF clock configuration.

`volatile uint32_t g_xtal0Freq`

External XTAL0 (OSC0/SYSOSC) clock frequency.

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

```
CLOCK_InitSysOsc(...);  
CLOCK_SetXtal0Freq(8000000);
```

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

`volatile uint32_t g_xtal32Freq`

External XTAL32/EXTAL32 clock frequency.

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

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

`static inline void CLOCK_EnableClock(clock_ip_name_t name)`

Enable the clock for specific IP.

Parameters

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

`static inline void CLOCK_EnableTPM2(void)`

Enable the TPM2 clock.

`static inline void CLOCK_EnableClockLPMMode(clock_ip_name_t name, clock_ip_control_t control)`

Enable the clock for specific IP in low power mode.

Parameters

- `name` – Which clock to enable, see `clock_ip_name_t`.
- `control` – Clock Config, see `clock_ip_control_t`.

`static inline void CLOCK_DisableClock(clock_ip_name_t name)`

Disable the clock for specific IP.

Parameters

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

`static inline void CLOCK_DisableTPM2(void)`

Disable the TPM2 clock.

`static inline void CLOCK_SetIpSrc(clock_ip_name_t name, clock_ip_src_t src)`

Set the clock source for specific IP module.

Set the clock source for specific IP, not all modules need to set the clock source, should only use this function for the modules need source setting. ERR052742: FRO6M clock is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

Parameters

- `name` – Which peripheral to check, see `clock_ip_name_t`.

- `src` – Clock source to set.

static inline void CLOCK_SetTpm2Src(*tpm2_src_t* src)

Set the clock source for TPM2.

Parameters

- `src` – Clock source to set.

static inline void CLOCK_SetIpSrcDiv(*clock_ip_name_t* name, uint8_t divValue)

Set the clock source and divider for specific IP module.

Set the clock source and divider for specific IP, not all modules need to set the clock source and divider, should only use this function for the modules need source and divider setting.

Divider output clock = Divider input clock / (divValue+1)).

Parameters

- `name` – Which peripheral to check, see *clock_ip_name_t*.
- `divValue` – The divider value.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in *clock_name_t*.

Parameters

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

Returns

Clock frequency value in hertz

uint32_t CLOCK_GetCoreSysClkFreq(void)

Get the core clock or system clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetPlatClkFreq(void)

Get the platform clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetBusClkFreq(void)

Get the bus clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFlashClkFreq(void)

Get the flash clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetIpFreq(*clock_ip_name_t* name)

Gets the functional clock frequency for a specific IP module.

This function gets the IP module's functional clock frequency based on MRCC registers. It is only used for the IP modules which could select clock source by MRCC[PCS].

Parameters

- `name` – Which peripheral to get, see *clock_ip_name_t*.

Returns

Clock frequency value in Hz

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

EDMA_CLOCKS

Clock ip name array for EDMA.

SYSPM_CLOCKS

Clock ip name array for SYSPM.

SFA_CLOCKS

Clock ip name array for SFA.

CRC_CLOCKS

Clock ip name array for CRC.

TPM_CLOCKS

Clock ip name array for TPM.

LPI2C_CLOCKS

Clock ip name array for LPI2C.

I3C_CLOCKS

Clock ip name array for I3C.

LPSPi_CLOCKS

Clock ip name array for LPSPi.

LPUART_CLOCKS

Clock ip name array for LPUART.

PORT_CLOCKS

Clock ip name array for PORT.

LPADC_CLOCKS

Clock ip name array for LPADC.

LPCMP_CLOCKS

Clock ip name array for LPCMP.

VREF_CLOCKS

Clock ip name array for VREF.

GPIO_CLOCKS

Clock ip name array for GPIO.

LPIT_CLOCKS

Clock ip name array for LPIT.

RF_CLOCKS

Clock ip name array for RF.

WDOG_CLOCKS

Clock ip name array for WDOG.

FLEXCAN_CLOCKS

Clock ip name array for FLEXCAN.

FLEXIO_CLOCKS

Clock ip name array for FLEXIO.

TSTMR_CLOCKS

Clock ip name array for TSTMR.

EWM_CLOCKS

Clock ip name array for EWM.

SEMA42_CLOCKS

Clock ip name array for SEMA42.

MU_CLOCKS

Clock ip name array for MU.

MAKE_MRCC_REGADDR(base, offset)

“IP Connector name definition used for clock gate, clock source and clock divider setting. It is defined as the corresponding register address.

CLOCK_REG(name)

struct _scg_sys_clk_config

#include <fsl_clock.h> SCG system clock configuration.

Public Members

uint32_t divSlow

Slow clock divider, see scg_sys_clk_div_t.

uint32_t divBus

Bus clock divider, see scg_sys_clk_div_t.

uint32_t __pad0__

Reserved.

uint32_t divCore

Core clock divider, see scg_sys_clk_div_t.

uint32_t __pad1__

Reserved.

uint32_t src

System clock source, see scg_sys_clk_src_t.

uint32_t __pad2__

reserved.

struct _scg_sosc_config

#include <fsl_clock.h> SCG system OSC configuration.

Public Members

uint32_t freq

System OSC frequency.

uint32_t enableMode

Enable mode, OR'ed value of _scg_sosc_enable_mode.

scg_sosc_monitor_mode_t monitorMode

Clock monitor mode selected.

struct _scg_rosc_config

#include <fsl_clock.h> SCG ROSC configuration.

Public Members

scg_rosc_monitor_mode_t monitorMode

Clock monitor mode selected.

struct _scg_sirc_config

#include <fsl_clock.h> SCG slow IRC clock configuration.

Public Members

scg_sirc_enable_mode_t enableMode

Enable mode, OR'ed value of _scg_sirc_enable_mode.

struct _scg_firc_trim_config

#include <fsl_clock.h> SCG fast IRC clock trim configuration.

Public Members

scg_firc_trim_mode_t trimMode

FIRC trim mode.

scg_firc_trim_src_t trimSrc

Trim source.

uint16_t trimDiv

Divider of SOSC for FIRC.

uint8_t trimCoar

Trim coarse value; Irrelevant if trimMode is kSCG_FircTrimUpdate.

uint8_t trimFine

Trim fine value; Irrelevant if trimMode is kSCG_FircTrimUpdate.

struct _scg_firc_config_t

#include <fsl_clock.h> SCG fast IRC clock configuration.

Public Members

uint32_t enableMode

Enable mode.

scg_firc_range_t range

Fast IRC frequency range.

const scg_firc_trim_config_t *trimConfig

Pointer to the FIRC trim configuration; set NULL to disable trim.

struct _fro192m_rf_clk_config

#include <fsl_clock.h> FRO192M RF clock configuration.

Public Members*fro192m_rf_range_t* range

FRO192M RF clock frequency range.

fro192m_rf_clk_div_t apb_rfcmc_div

RF Flash APB and RF_CMC clock divide.

2.10 CMC: Core Mode Controller Driver

`void CMC_SetClockMode(CMC_Type *base, cmc_clock_mode_t mode)`

Sets clock mode.

This function config the amount of clock gating when the core asserts Sleeping due to WFI, WFE or SLEEPONEXIT.

Parameters

- base – CMC peripheral base address.
- mode – System clock mode.

`static inline void CMC_LockClockModeSetting(CMC_Type *base)`

Locks the clock mode setting.

After invoking this function, any clock mode setting will be blocked.

Parameters

- base – CMC peripheral base address.

`static inline cmc_core_clock_gate_status_t CMC_GetCoreClockGatedStatus(CMC_Type *base)`

Gets the core clock gated status.

This function get the status to indicate whether the core clock is gated. The core clock gated status can be cleared by software.

Parameters

- base – CMC peripheral base address.

Returns

The status to indicate whether the core clock is gated.

`static inline void CMC_ClearCoreClockGatedStatus(CMC_Type *base)`

Clears the core clock gated status.

This function clear clock status flag by software.

Parameters

- base – CMC peripheral base address.

`static inline uint8_t CMC_GetWakeupSource(CMC_Type *base)`

Gets the Wakeup Source.

This function gets the Wakeup sources from the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns

The Wakeup sources from the previous low power mode entry. See `_cmc_wakeup_sources` for details.

static inline *cmc_clock_mode_t* CMC_GetClockMode(CMC_Type *base)

Gets the Clock mode.

This function gets the clock mode of the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns

The Low Power status.

static inline uint32_t CMC_GetSystemResetStatus(CMC_Type *base)

Gets the System reset status.

This function returns the system reset status. Those status updates on every MAIN Warm Reset to indicate the type/source of the most recent reset.

Parameters

- base – CMC peripheral base address.

Returns

The most recent system reset status. See `_cmc_system_reset_sources` for details.

static inline uint32_t CMC_GetStickySystemResetStatus(CMC_Type *base)

Gets the sticky system reset status since the last WAKE Cold Reset.

This function gets all source of system reset that have generated a system reset since the last WAKE Cold Reset, and that have not been cleared by software.

Parameters

- base – CMC peripheral base address.

Returns

System reset status that have not been cleared by software. See `_cmc_system_reset_sources` for details.

static inline void CMC_ClearStickySystemResetStatus(CMC_Type *base, uint32_t mask)

Clears the sticky system reset status flags.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the sticky system reset status to be cleared.

static inline uint8_t CMC_GetResetCount(CMC_Type *base)

Gets the number of reset sequences completed since the last WAKE Cold Reset.

Parameters

- base – CMC peripheral base address.

Returns

The number of reset sequences.

static inline uint32_t CMC_GetResetInitStatusFlags(CMC_Type *base)

Gets status flags to indicate if any errors occurred during the reset initialization sequence.

Parameters

- base – CMC peripheral base address.

Returns

Status flags that indicate errors occurred during the reset initialization sequence.

```
void CMC_SetPowerModeProtection(CMC_Type *base, uint32_t allowedModes)
```

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes. This should be done before set the lowPower mode for each power doamin.

The allowed lowpower modes are passed as bit map. For example, to allow Sleep and DeepSleep, use `CMC_SetPowerModeProtection(CMC_base, kCMC_AllowSleepMode | kCMC_AllowDeepSleepMode)`. To allow all low power modes, use `CMC_SetPowerModeProtection(CMC_base, kCMC_AllowAllLowPowerModes)`.

Parameters

- `base` – CMC peripheral base address.
- `allowedModes` – Bitmaps of the allowed power modes. See `_cmc_power_mode_protection` for details.

```
static inline void CMC_LockPowerModeProtectionSetting(CMC_Type *base)
```

Locks the power mode protection.

This function locks the power mode protection. After invoking this function, any power mode protection setting will be ignored.

Parameters

- `base` – CMC peripheral base address.

```
static inline void CMC_SetGlobalPowerMode(CMC_Type *base, cmc_low_power_mode_t  
lowPowerMode)
```

Config the same lowPower mode for all power domain.

This function configures the same low power mode for MAIN power domian and WAKE power domain.

Parameters

- `base` – CMC peripheral base address.
- `lowPowerMode` – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

```
static inline void CMC_SetMAINPowerMode(CMC_Type *base, cmc_low_power_mode_t  
lowPowerMode)
```

Configures entry into low power mode for the MAIN Power domain.

This function configures the low power mode for the MAIN power domian, when the core executes WFI/WFE instruction. The available lowPower modes are defined in the `cmc_low_power_mode_t`.

Parameters

- `base` – CMC peripheral base address.
- `lowPowerMode` – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

```
static inline cmc_low_power_mode_t CMC_GetMAINPowerMode(CMC_Type *base)
```

Gets the power mode of the MAIN Power domain.

Parameters

- `base` – CMC peripheral base address.

Returns

The power mode of MAIN Power domain. See `cmc_low_power_mode_t` for details.

```
static inline void CMC_SetWAKEPowerMode(CMC_Type *base, cmc_low_power_mode_t
                                         lowPowerMode)
```

Configure entry into low power mode for the WAKE Power domain.

This function configures the low power mode for the WAKE power domain, when the core executes WFI/WFE instruction. The available lowPower mode are defined in the *cmc_low_power_mode_t*.

Note: The lowPower Mode for the WAKE domain must not be configured to a lower power mode than any other power domain.

Parameters

- *base* – CMC peripheral base address.
- *lowPowerMode* – The desired lowPower mode. See *cmc_low_power_mode_t* for details.

```
static inline cmc_low_power_mode_t CMC_GetWAKEPowerMode(CMC_Type *base)
```

Gets the power mode of the WAKE Power domain.

Parameters

- *base* – CMC peripheral base address.

Returns

The power mode of WAKE Power domain. See *cmc_low_power_mode_t* for details.

```
void CMC_ConfigResetPin(CMC_Type *base, const cmc_reset_pin_config_t *config)
```

Configure reset pin.

This function configures reset pin. When enabled, the low power filter is enabled in both Active and Low power modes, the reset filter is only enabled in Active mode. When both filters are enabled, they operate in series.

Parameters

- *base* – CMC peripheral base address.
- *config* – Pointer to the reset pin config structure.

```
static inline void CMC_EnableSystemResetInterrupt(CMC_Type *base, uint32_t mask)
```

Enable system reset interrupts.

This function enables the system reset interrupts. The assertion of non-fatal warm reset can be delayed for 258 cycles of the 32K_CLK clock while an enabled interrupt is generated. Then Software can perform a graceful shutdown or abort the non-fatal warm reset provided the pending reset source is cleared by resetting the reset source and then clearing the pending flag.

Parameters

- *base* – CMC peripheral base address.
- *mask* – System reset interrupts. See *_cmc_system_reset_interrupt_enable* for details.

```
static inline void CMC_DisableSystemResetInterrupt(CMC_Type *base, uint32_t mask)
```

Disable system reset interrupts.

This function disables the system reset interrupts.

Parameters

- *base* – CMC peripheral base address.

- mask – System reset interrupts. See `_cmc_system_reset_interrupt_enable` for details.

static inline uint32_t CMC_GetSystemResetInterruptFlags(CMC_Type *base)

Gets System Reset interrupt flags.

This function returns the System reset interrupt flags.

Parameters

- base – CMC peripheral base address.

Returns

System reset interrupt flags. See `_cmc_system_reset_interrupt_flag` for details.

static inline void CMC_ClearSystemResetInterruptFlags(CMC_Type *base, uint32_t mask)

Clears System Reset interrupt flags.

This function clears system reset interrupt flags. The pending reset source can be cleared by resetting the source of the reset and then clearing the pending flags.

Parameters

- base – CMC peripheral base address.
- mask – System Reset interrupt flags. See `_cmc_system_reset_interrupt_flag` for details.

static inline void CMC_EnableNonMaskablePinInterrupt(CMC_Type *base, bool enable)

Enable/Disable Non maskable Pin interrupt.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Non maskable pin interrupt. true - enable Non-maskable pin interrupt. false - disable Non-maskable pin interrupt.

static inline uint8_t CMC_GetISPMODEPinLogic(CMC_Type *base)

Gets the logic state of the ISPMODE_n pin.

This function returns the logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

Parameters

- base – CMC peripheral base address.

Returns

The logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

static inline void CMC_ClearISPMODEPinLogic(CMC_Type *base)

Clears ISPMODE_n pin state.

Parameters

- base – CMC peripheral base address.

static inline void CMC_ForceBootConfiguration(CMC_Type *base, bool assert)

Set the logic state of the BOOT_CONFIGn pin.

This function force the logic state of the Boot_Confign pin to assert on next system reset.

Parameters

- base – CMC peripheral base address.
- assert – Assert the corresponding pin or not. true - Assert corresponding pin on next system reset. false - No effect.

```
static inline void CMC_LockWriteOperationToBootRomStatusReg(CMC_Type *base, uint8_t index)
```

Lock write operation to BootROM status register and BootROM Lock register.

Note: If locked, BootROM status register cannot be written.

Note: Once locked, only cold reset can reset related register.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

```
static inline bool CMC_CheckBootRomStatusRegWriteLocked(CMC_Type *base, uint8_t index)
```

Check if BootROM status register can be written.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Return values

- true – The selected BootRom status register is locked and cannot be written.
- false – The selected BootRom Status register is unlocked and cannot be written.

```
static inline uint32_t CMC_GetBootRomStatus(CMC_Type *base, uint8_t index)
```

Gets the information written by the BootROM.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Returns

The status information written by the BootROM.

```
static inline void CMC_WriteBootRomStatusReg(CMC_Type *base, uint8_t index, uint32_t value)
```

Writes value to BootROM status register, in this way, BootROM status registers are used as general purpose register.

Note: Value in BootROM status registers are reset in cold reset.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.
- value – Value to write.

```
void CMC_PowerOffSRAMAllMode(CMC_Type *base, uint32_t mask)
```

Power off the selected system SRAM always.

This function power off the selected system SRAM always. The SRAM arrays should not be accessed while they are shut down. SRAM array contents are not retained if they are powered off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered off all modes. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
static inline void CMC_PowerOnSRAMAllMode(CMC_Type *base, uint32_t mask)
```

Power on SRAM during all mode.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered on all modes. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
void CMC_PowerOffSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)
```

Power off the selected system SRAM during low power mode only.

This function power off the selected system SRAM only during low power mode. SRAM array contents are not retained if they are power off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power off during low power mode only. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
static inline void CMC_PowerOnSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)
```

Power on the selected system SRAM during low power mode only.

This function power on the selected system SRAM. The SRAM array contents are retained in low power modes.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power on during low power mode only. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
void CMC_ConfigFlashMode(CMC_Type *base, bool wake, bool doze, bool disable)
```

Configs the low power mode of the on-chip flash memory.

This function configs the low power mode of the on-chip flash memory.

Parameters

- base – CMC peripheral base address.
- wake – true: Flash will exit low power state during the flash memory accesses. false: No effect.
- doze – true: Flash is disabled while core is sleeping false: No effect.
- disable – true: Flash memory is placed in low power state. false: No effect.

```
static inline void CMC_EnableDebugOperation(CMC_Type *base, bool enable)
```

Enables/Disables debug Operation when the core sleep.

This function configs what happens to debug when core sleeps.

Parameters

- `base` – CMC peripheral base address.
- `enable` – Enable or disable Debug when Core is sleeping. `true` - Debug remains enabled when the core is sleeping. `false` - Debug is disabled when the core is sleeping.

`void CMC_PreEnterLowPowerMode(void)`

Prepares to enter low power modes.

This function should be called before entering low power modes.

`void CMC_PostExitLowPowerMode(void)`

Recovers after wake up from low power modes.

This function should be called after wake up from low power modes. This function should be used with `CMC_PreEnterLowPowerMode()`

`void CMC_GlobalEnterLowPowerMode(CMC_Type *base, cmc_low_power_mode_t lowPowerMode)`

Configures the entry into the same low power mode for each power domains.

This function provides the feature to entry into the same low power mode for each power domains. Before invoking this function, please ensure the selected power mode have been allowed.

Parameters

- `base` – CMC peripheral base address.
- `lowPowerMode` – The low power mode to be entered. See `cmc_low_power_mode_t` for the details.

`void CMC_EnterLowPowerMode(CMC_Type *base, const cmc_power_domain_config_t *config)`

Configures the entry into different low power modes for each power domains.

This function provides the feature to entry into different low power modes for each power domains. Before invoking this function please ensure the selected modes are allowed.

Parameters

- `base` – CMC peripheral base address.
- `config` – Pointer to the `cmc_power_domain_config_t` structure.

`FSL_CMC_DRIVER_VERSION`

CMC driver version 2.4.4.

`CMC_SRAM_BUSY_TIMEOUT`

Max loops to wait for CMC SRAM operation complete.

When configuring the SRAM, driver will wait for the completion of new settings. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

`enum _cmc_power_mode_protection`

CMC power mode Protection enumeration.

Values:

enumerator `kCMC_AllowSleepMode`

Allow Sleep mode.

enumerator `kCMC_AllowDeepSleepMode`

Allow Deep Sleep mode.

enumerator `kCMC_AllowPowerDownMode`

Allow Power Down mode.

enumerator kCMC_AllowDeepPowerDownMode
Allow Deep Power Down mode.

enumerator kCMC_AllowAllLowPowerModes
Allow all low power modes.

enum __cmc_wakeup_sources

Wake up sources from the previous low power mode entry.

Values:

enumerator kCMC_WakeupFromResetInterruptOrPowerDown
Wakeup source is reset interrupt, or wake up from [Deep] Power Down.

enumerator kCMC_WakeupFromDebugRequest
Wakeup source is debug request.

enumerator kCMC_WakeupFromInterrupt
Wakeup source is interrupt.

enumerator kCMC_WakeupFromDMAWakeup
Wakeup source is DMA Wakeup.

enumerator kCMC_WakeupFromWUURequest
Wakeup source is WUU request.

enumerator kCMC_WakeupFromBusMaster
Wakeup source is Bus master.

enum __cmc_system_reset_interrupt_enable

System Reset Interrupt enable enumeration.

Values:

enumerator kCMC_PinResetInterruptEnable
Pin Reset interrupt enable.

enumerator kCMC_DAPResetInterruptEnable
DAP Reset interrupt enable.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetInterruptEnable
Low Power Acknowledge Timeout Reset interrupt enable.

enumerator kCMC_SystemClockGenerationResetInterruptEnable
System Clock Generation Reset interrupt enable.

enumerator kCMC_Watchdog0ResetInterruptEnable
Watchdog 0 Reset interrupt enable.

enumerator kCMC_SoftwareResetInterruptEnable
Software Reset interrupt enable.

enumerator kCMC_LockupResetInterruptEnable
Lockup Reset interrupt enable.

enumerator kCMC_Watchdog1ResetInterruptEnable
Watchdog 1 Reset interrupt enable

enum __cmc_system_reset_interrupt_flag

CMC System Reset Interrupt Status flag.

Values:

enumerator kCMC_PinResetInterruptFlag

Pin Reset interrupt flag.

enumerator kCMC_DAPResetInterruptFlag

DAP Reset interrupt flag.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetFlag

Low Power Acknowledge Timeout Reset interrupt flag.

enumerator kCMC_Watchdog0ResetInterruptFlag

Watchdog 0 Reset interrupt flag.

enumerator kCMC_SoftwareResetInterruptFlag

Software Reset interrupt flag.

enumerator kCMC_LockupResetInterruptFlag

Lock up Reset interrupt flag.

enumerator kCMC_Watchdog1ResetInterruptFlag

Watchdog 1 Reset interrupt flag.

enum _cmc_system_sram_arrays

CMC System SRAM arrays low power mode enable enumeration.

Values:

enumerator kCMC_SRAMBank0

Power off SRAM Bank0, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank1

Power off SRAM Bank1, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank2

Power off SRAM Bank2, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank3

Power off SRAM Bank3, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank4

Power off SRAM Bank4, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank5

Power off SRAM Bank5, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank6

Power off SRAM Bank6, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank7

Power off SRAM Bank7, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank8

Power off SRAM Bank8, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank9

Power off SRAM Bank9, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank10

Power off SRAM Bank10, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_AllSramArrays

Mask of all system SRAM arrays.

enum _cmc_system_reset_sources

System reset sources enumeration.

Values:

enumerator kCMC_WakeUpReset

The reset caused by a wakeup from Power Down or Deep Power Down mode.

enumerator kCMC_PORReset

The reset caused by power on reset detection logic.

enumerator kCMC_LVDRReset

The reset caused by a Low Voltage Detect.

enumerator kCMC_HVDRReset

The reset caused by a High voltage Detect.

enumerator kCMC_WarmReset

The last reset source is a warm reset source.

enumerator kCMC_FatalReset

The last reset source is a fatal reset source.

enumerator kCMC_PinReset

The reset caused by the RESET_b pin.

enumerator kCMC_DAPReset

The reset caused by a reset request from the Debug Access port.

enumerator kCMC_ResetTimeout

The reset caused by a timeout or other error condition in the system reset generation.

enumerator kCMC_LowPowerAcknowledgeTimeoutReset

The reset caused by a timeout in low power mode entry logic.

enumerator kCMC_SCGRReset

The reset caused by a loss of clock or loss of lock event in the SCG.

enumerator kCMC_Watchdog0Reset

The reset caused by a WatchDog 0 timeout.

enumerator kCMC_SoftwareReset

The reset caused by a software reset request.

enumerator kCMC_LockUpReset

The reset caused by the ARM core indication of a LOCKUP event.

enumerator kCMC_Watchdog1Reset

The reset caused by a WatchDog 1 timeout.

enum _cmc_core_clock_gate_status

Indicate the core clock was gated.

Values:

enumerator kCMC_CoreClockNotGated

Core clock not gated.

enumerator kCMC_CoreClockGated

Core clock was gated due to low power mode entry.

enum _cmc_clock_mode

CMC clock mode enumeration.

Values:

enumerator kCMC_GateNoneClock
No clock gating.

enumerator kCMC_GateCoreClock
Gate Core clock.

enumerator kCMC_GateCorePlatformClock
Gate Core clock and platform clock.

enumerator kCMC_GateAllSystemClocks
Gate all System clocks, without getting core entering into low power mode.

enumerator kCMC_GateAllSystemClocksEnterLowPowerMode
Gate all System clocks, with core entering into low power mode.

enum _cmc_low_power_mode
CMC power mode enumeration.
Values:

enumerator kCMC_ActiveMode
Select Active mode.

enumerator kCMC_SleepMode
Select Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepSleepMode
Select Deep Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_PowerDownMode
Select Power Down mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepPowerDown
Select Deep Power Down mode when a core executes WFI or WFE instruction.

typedef enum _cmc_core_clock_gate_status cmc_core_clock_gate_status_t
Indicate the core clock was gated.

typedef enum _cmc_clock_mode cmc_clock_mode_t
CMC clock mode enumeration.

typedef enum _cmc_low_power_mode cmc_low_power_mode_t
CMC power mode enumeration.

typedef struct _cmc_reset_pin_config cmc_reset_pin_config_t
CMC reset pin configuration.

typedef struct _cmc_power_domain_config cmc_power_domain_config_t
power mode configuration for each power domain.

CMC_BLR_LOCK_FIELD_WIDTH

CMC_BLR_LOCK_IDX_MASK(index)

CMC_BLR_LOCK_IDX_SHIFT(index)

CMC_BLR_LOCK_IDX(index, value)

struct _cmc_reset_pin_config
#include <fsl_cmc.h> CMC reset pin configuration.

Public Members

bool lowpowerFilterEnable

Low Power Filter enable.

bool resetFilterEnable

Reset Filter enable.

uint8_t resetFilterWidth

Width of the Reset Filter.

struct __cmc_power_domain_config

#include <fsl_cmc.h> power mode configuration for each power domain.

Public Members

cmc_clock_mode_t clock_mode

Clock mode for each power domain.

cmc_low_power_mode_t main_domain

The low power mode of the MAIN power domain.

cmc_low_power_mode_t wake_domain

The low power mode of the WAKE power domain.

2.11 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()`

2.12 EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

`void EDMA_Init(DMA_Type *base, const edma_config_t *config)`

Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Note: This function enables the minor loop map feature.

Parameters

- `base` – eDMA peripheral base address.
- `config` – A pointer to the configuration structure, see “`edma_config_t`”.

`void EDMA_Deinit(DMA_Type *base)`

Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- `base` – eDMA peripheral base address.

`void EDMA_InstallTCD(DMA_Type *base, uint32_t channel, edma_tcd_t *tcd)`

Push content of TCD structure into hardware TCD register.

Parameters

- `base` – EDMA peripheral base address.
- `channel` – EDMA channel number.
- `tcd` – Point to TCD structure.

`void EDMA_GetDefaultConfig(edma_config_t *config)`

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values:

```
config.enableMasterIdReplication = true;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
config.enableBufferedWrites = false;
```

Parameters

- `config` – A pointer to the eDMA configuration structure.

static inline void EDMA_EnableAllChannelLink(DMA_Type *base, bool enable)

Enables/disables all channel linking.

This function enables/disables all channel linking in the management page. For specific channel linking enablement & configuration, please refer to `EDMA_SetChannelLink` and `EDMA_TcdSetChannelLink` APIs.

For example, to disable all channel linking in the DMA0 management page:

```
EDMA_EnableAllChannelLink(DMA0, false);
```

Parameters

- `base` – eDMA peripheral base address.
- `enable` – Switcher of the channel linking feature for all channels. “true” means to enable. “false” means not.

void EDMA_ResetChannel(DMA_Type *base, uint32_t channel)

Sets all TCD registers to default values.

This function sets TCD registers for this channel to default values.

Note: This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

Note: This function enables the auto stop request feature.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.

void EDMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *edma_transfer_config_t* *config, *edma_tcd_t* *nextTcd)

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_config_t config;
edma_tcd_t tcd;
config.srcAddr = ..;
config.destAddr = ..;
...
EDMA_SetTransferConfig(DMA0, channel, &config, &stcd);
```

Note: If `nextTcd` is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the `EDMA_ResetChannel`.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.

- `config` – Pointer to eDMA transfer configuration structure.
- `nextTcd` – Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA__SetMinorOffsetConfig(DMA_Type *base, uint32_t channel, const
                                edma_minor_offset_config_t *config)
```

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.
- `config` – A pointer to the minor offset configuration structure.

```
static inline void EDMA__SetChannelArbitrationGroup(DMA_Type *base, uint32_t channel,
                                                    uint32_t group)
```

Configures the eDMA channel arbitration group.

This function configures the channel arbitration group. The arbitration group priorities are evaluated by numeric value from highest group number to lowest.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number
- `group` – Fixed-priority arbitration group number for the channel.

```
static inline void EDMA__SetChannelPreemptionConfig(DMA_Type *base, uint32_t channel, const
                                                    edma_channel_preemption_config_t
                                                    *config)
```

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number
- `config` – A pointer to the channel preemption configuration structure.

```
static inline uint32_t EDMA__GetChannelSystemBusInformation(DMA_Type *base, uint32_t
                                                           channel)
```

Gets the eDMA channel identification and attribute information on the system bus interface.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.

Returns

The mask of the channel system bus information. Users need to use the `_edma_channel_sys_bus_info` type to decode the return variables.

```
void EDMA__SetChannelLink(DMA_Type *base, uint32_t channel, edma_channel_link_type_t
                          type, uint32_t linkedChannel)
```

Sets the channel link for the eDMA transfer.

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- type – A channel link type, which can be one of the following:
 - kEDMA_LinkNone
 - kEDMA_MinorLink
 - kEDMA_MajorLink
- linkedChannel – The linked channel number.

void EDMA_SetBandWidth(DMA_Type *base, uint32_t channel, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA transfer.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- bandWidth – A bandwidth setting, which can be one of the following:
 - kEDMABandwidthStallNone
 - kEDMABandwidthStall4Cycle
 - kEDMABandwidthStall8Cycle

void EDMA_SetModulo(DMA_Type *base, uint32_t channel, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA transfer.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

```
static inline void EDMA__EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)
```

Enables an async request for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

```
static inline void EDMA__EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)
```

Enables an auto stop request for the eDMA transfer.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

```
void EDMA__EnableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Enables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

```
void EDMA__DisableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Disables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of the interrupt source to be set. Use the defined `edma_interrupt_enable_t` type.

```
static inline void EDMA__SetChannelMux(DMA_Type *base, uint32_t channel, uint32_t mux)
```

Set channel mux source.

Note:When the peripheral is no longer needed, the mux configuration for that channel should be written to 0, thus releasing the resource.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mux – the mux source value is SOC specific, please reference the SOC for detail.

```
void EDMA__TcdReset(edma_tcd_t *tcd)
```

Sets all fields to default values for the TCD structure.

This function sets all fields for this TCD structure to default value.

Note: This function enables the auto stop request feature.

Parameters

- `tcd` – Pointer to the TCD structure.

```
void EDMA_TcdSetTransferConfig(edma_tcd_t *tcd, const edma_transfer_config_t *config,
                             edma_tcd_t *nextTcd)
```

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_config_t config = {
...
}
edma_tcd_t tcd __aligned(32);
edma_tcd_t nextTcd __aligned(32);
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the `nextTcd` is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the `EDMA_TcdReset`.

Parameters

- `tcd` – Pointer to the TCD structure.
- `config` – Pointer to eDMA transfer configuration structure.
- `nextTcd` – Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA_TcdSetMinorOffsetConfig(edma_tcd_t *tcd, const edma_minor_offset_config_t
                                 *config)
```

Configures the eDMA TCD minor offset feature.

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

- `tcd` – A point to the TCD structure.
- `config` – A pointer to the minor offset configuration structure.

```
void EDMA_TcdSetChannelLink(edma_tcd_t *tcd, edma_channel_link_type_t type, uint32_t
                           linkedChannel)
```

Sets the channel link for the eDMA TCD.

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- `tcd` – Point to the TCD structure.
- `type` – Channel link type, it can be one of:
 - `kEDMA_LinkNone`
 - `kEDMA_MinorLink`
 - `kEDMA_MajorLink`
- `linkedChannel` – The linked channel number.

static inline void EDMA_TcdSetBandWidth(*edma_tcd_t* *tcd, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA TCD.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- `tcd` – A pointer to the TCD structure.
- `bandWidth` – A bandwidth setting, which can be one of the following:
 - `kEDMABandwidthStallNone`
 - `kEDMABandwidthStall4Cycle`
 - `kEDMABandwidthStall8Cycle`

void EDMA_TcdSetModulo(*edma_tcd_t* *tcd, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA TCD.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- `tcd` – A pointer to the TCD structure.
- `srcModulo` – A source modulo value.
- `destModulo` – A destination modulo value.

static inline void EDMA_TcdEnableAutoStopRequest(*edma_tcd_t* *tcd, bool enable)

Sets the auto stop request for the eDMA TCD.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- `tcd` – A pointer to the TCD structure.
- `enable` – The command to enable (true) or disable (false).

void EDMA__TcdEnableInterrupts(*edma_tcd_t* *tcd, uint32_t mask)

Enables the interrupt source for the eDMA TCD.

Parameters

- tcd – Point to the TCD structure.
- mask – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

void EDMA__TcdDisableInterrupts(*edma_tcd_t* *tcd, uint32_t mask)

Disables the interrupt source for the eDMA TCD.

Parameters

- tcd – Point to the TCD structure.
- mask – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

static inline void EDMA__EnableChannelRequest(DMA_Type *base, uint32_t channel)

Enables the eDMA hardware channel request.

This function enables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

static inline void EDMA__DisableChannelRequest(DMA_Type *base, uint32_t channel)

Disables the eDMA hardware channel request.

This function disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

static inline void EDMA__TriggerChannelStart(DMA_Type *base, uint32_t channel)

Starts the eDMA transfer by using the software trigger.

This function starts a minor loop transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

uint32_t EDMA__GetRemainingMajorLoopCount(DMA_Type *base, uint32_t channel)

Gets the Remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.

- a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in `DMA_TCDn_NBYTES_MLNO` register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the `dma_engine` is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling

the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTES(initially configured)

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

Returns

Major loop count which has not been transferred yet for the current TCD.

```
static inline uint32_t EDMA_GetErrorStatusFlags(DMA_Type *base)
```

Gets the eDMA channel error status flags.

Parameters

- base – eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the `_edma_error_status_flags` type to decode the return variables.

```
uint32_t EDMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)
```

Gets the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

Returns

The mask of channel status flags. Users need to use the `_edma_channel_status_flags` type to decode the return variables.

```
void EDMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Clears the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of channel status to be cleared. Users need to use the defined `_edma_channel_status_flags` type.

```
void EDMA_CreateHandle(edma_handle_t *handle, DMA_Type *base, uint32_t channel)
```

Creates the eDMA handle.

This function is called if using the transactional API for eDMA. This function initializes the internal state of the eDMA handle.

Parameters

- handle – eDMA handle pointer. The eDMA handle stores callback function and parameters.
- base – eDMA peripheral base address.
- channel – eDMA channel number.

```
void EDMA_InstallTCDMemory(edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
```

Installs the TCDs memory pool into the eDMA handle.

This function is called after the `EDMA_CreateHandle` to use scatter/gather feature.

Parameters

- handle – eDMA handle pointer.
- tedPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- tedSize – The number of TCD slots.

void EDMA_SetCallback(*edma_handle_t* *handle, *edma_callback* callback, void *userData)

Installs a callback function for the eDMA transfer.

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

void EDMA_PrepareTransferConfig(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type *edma_transfer_config_t*.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- srcOffset – eDMA transfer source address offset
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- destOffset – eDMA transfer destination address offset
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.

void EDMA_PrepareTransfer(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, *edma_transfer_type_t* transferType)

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- `config` – The user configuration structure of type `edma_transfer_config_t`.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.
- `transferType` – eDMA transfer type.

`status_t` EDMA_SubmitTransfer(*edma_handle_t* *handle, const *edma_transfer_config_t* *config)

Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- `handle` – eDMA handle pointer.
- `config` – Pointer to eDMA transfer configuration structure.

Return values

- `kStatus_EDMA_Success` – It means submit transfer request succeed.
- `kStatus_EDMA_QueueFull` – It means TCD queue is full. Submit transfer request is not allowed.
- `kStatus_EDMA_Busy` – It means the given channel is busy, need to submit request later.

`void` EDMA_StartTransfer(*edma_handle_t* *handle)

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- `handle` – eDMA handle pointer.

`void` EDMA_StopTransfer(*edma_handle_t* *handle)

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call `EDMA_StartTransfer()` again to resume the transfer.

Parameters

- `handle` – eDMA handle pointer.

`void` EDMA_AbortTransfer(*edma_handle_t* *handle)

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- `handle` – DMA handle pointer.

static inline uint32_t EDMA_GetUnusedTCDNumber(*edma_handle_t* *handle)

Get unused TCD slot number.

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

- handle – DMA handle pointer.

Returns

The unused tcd slot number.

static inline uint32_t EDMA_GetNextTCDAddress(*edma_handle_t* *handle)

Get the next tcd address.

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

- handle – DMA handle pointer.

Returns

The next TCD address.

static inline *edma_transfer_size_t* EDMA_GetTransferSize(uint32_t width)

Get the transfer size.

This function gets the transfer size.

Parameters

- width – transfer width(bytes).

Returns

The transfer size.

void EDMA_HandleIRQ(*edma_handle_t* *handle)

eDMA IRQ handler for the current major loop transfer completion.

This function clears the channel major interrupt flag and calls the callback function if it is not NULL.

Parameters

- handle – eDMA handle pointer.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.5.3.

enum *_edma_transfer_size*

eDMA transfer configuration

Values:

enumerator kEDMA_TransferSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kEDMA_TransferSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kEDMA_TransferSize4Bytes

Source/Destination data transfer size is 4 bytes every time

enumerator kEDMA_TransferSize8Bytes

Source/Destination data transfer size is 8 bytes every time

enumerator kEDMA_TransferSize16Bytes

Source/Destination data transfer size is 16 bytes every time

enumerator kEDMA_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enumerator kEDMA_TransferSize64Bytes

Source/Destination data transfer size is 64 bytes every time

enum __edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes

Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes

Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes

Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes

Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes

Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes

Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes

Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes

Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes

Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes

Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes

Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes

Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes

Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes

Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes

Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes

Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes

Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes

Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes

Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes

Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes

Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes

Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes

Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes

Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes

Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes

Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes

Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes

Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes

Circular buffer size is 2 G bytes.

enum __edma_bandwidth

Bandwidth control.

Values:

enumerator kEDMA_BandwidthStallNone

No eDMA engine stalls.

enumerator kEDMA_BandwidthStall4Cycle

eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_BandwidthStall8Cycle

eDMA engine stalls for 8 cycles after each read/write.

enum _edma_channel_link_type

Channel link type.

Values:

enumerator kEDMA_LinkNone

No channel link

enumerator kEDMA_MinorLink

Channel link after each minor loop

enumerator kEDMA_MajorLink

Channel link while major loop count exhausted

eDMA channel status flags, _edma_channel_status_flags

Values:

enumerator kEDMA_DoneFlag

DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag

eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag

eDMA interrupt flag, set while an interrupt occurred of this channel

eDMA channel error status flags, _edma_error_status_flags

Values:

enumerator kEDMA_DestinationBusErrorFlag

Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag

Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag

Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag

NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag

Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag

Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag

Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag

Source address not aligned with source size

enumerator kEDMA_TransferCanceledFlag

Transfer cancelled

enumerator kEDMA_ErrorChannelFlag

Error channel number of the cancelled channel number

enumerator kEDMA_ValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

eDMA channel system bus information, _edma_channel_sys_bus_info

Values:

enumerator kEDMA_AttributeOutput

DMA's AHB system bus attribute output value.

enumerator kEDMA_PrivilegedAccessLevel

Privileged Access Level for DMA transfers. 0b - User protection level; 1b - Privileged protection level.

enumerator kEDMA_MasterId

DMA's master ID when channel is active and master ID replication is enabled.

enum _edma_interrupt_enable

eDMA interrupt source

Values:

enumerator kEDMA_ErrorInterruptEnable

Enable interrupt while channel error occurs.

enumerator kEDMA_MajorInterruptEnable

Enable interrupt while major count exhausted.

enumerator kEDMA_HalfInterruptEnable

Enable interrupt while major count to half value.

enum _edma_transfer_type

eDMA transfer type

Values:

enumerator kEDMA_MemoryToMemory

Transfer from memory to memory

enumerator kEDMA_PeripheralToMemory

Transfer from peripheral to memory

enumerator kEDMA_MemoryToPeripheral

Transfer from memory to peripheral

enumerator kEDMA_PeripheralToPeripheral

Transfer from Peripheral to peripheral

eDMA transfer status, _edma_transfer_status

Values:

enumerator kStatus_EDMA_QueueFull

TCD queue is full.

enumerator kStatus_EDMA_Busy

Channel is busy and can't handle the transfer request.

typedef enum _edma_transfer_size edma_transfer_size_t

eDMA transfer configuration

typedef enum *_edma_modulo* edma__modulo__t
eDMA modulo configuration

typedef enum *_edma_bandwidth* edma__bandwidth__t
Bandwidth control.

typedef enum *_edma_channel_link_type* edma__channel__link__type__t
Channel link type.

typedef enum *_edma_interrupt_enable* edma__interrupt__enable__t
eDMA interrupt source

typedef enum *_edma_transfer_type* edma__transfer__type__t
eDMA transfer type

typedef struct *_edma_config* edma__config__t
eDMA global configuration structure.

typedef struct *_edma_transfer_config* edma__transfer__config__t
eDMA transfer configuration

This structure configures the source/destination transfer attribute.

typedef struct *_edma_channel_Preemption_config* edma__channel__Preemption__config__t
eDMA channel priority configuration

typedef struct *_edma_minor_offset_config* edma__minor__offset__config__t
eDMA minor offset configuration

typedef struct *_edma_tcd* edma__tcd__t
eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

typedef void (*edma__callback)(struct *_edma_handle* *handle, void *userData, bool transferDone, uint32_t tcDs)

Define callback function for eDMA.

typedef uint32_t (*edma__memorymap_callback)(uint32_t addr)
Memroy map function callback for DMA.

typedef struct *_edma_handle* edma__handle__t
eDMA transfer handle structure

struct *_edma_config*
#include <fsl_edma.h> eDMA global configuration structure.

Public Members

bool enableMasterIdReplication
Enable (true) master ID replication. If Master ID replication is disabled, the privileged protection level (supervisor mode) for DMA transfers is used.

bool enableHaltOnError
Enable (true) transfer halt on error. Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

bool enableRoundRobinArbitration
Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection

bool enableDebugMode

Enable(true) eDMA debug mode. When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

struct __edma__transfer__config

#include <fsl_edma.h> eDMA transfer configuration

This structure configures the source/destination transfer attribute.

Public Members

uint32_t srcAddr

Source data address.

uint32_t destAddr

Destination data address.

edma_transfer_size_t srcTransferSize

Source data transfer size.

edma_transfer_size_t destTransferSize

Destination data transfer size.

int16_t srcOffset

Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.

int16_t destOffset

Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.

uint32_t minorLoopBytes

Bytes to transfer in a minor loop

uint32_t majorLoopCounts

Major loop iteration count.

struct __edma__channel__Preemption__config

#include <fsl_edma.h> eDMA channel priority configuration

Public Members

bool enableChannelPreemption

If true: a channel can be suspended by other channel with higher priority

bool enablePreemptAbility

If true: a channel can suspend other channel with low priority

uint8_t channelPriority

Channel priority

struct __edma__minor__offset__config

#include <fsl_edma.h> eDMA minor offset configuration

Public Members

bool enableSrcMinorOffset

Enable(true) or Disable(false) source minor loop offset.

bool enableDestMinorOffset

Enable(true) or Disable(false) destination minor loop offset.

uint32_t minorOffset

Offset for a minor loop mapping.

struct _edma_tcd

#include <fsl_edma.h> eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

Public Members

__IO uint32_t SADDR

SADDR register, used to save source address

__IO uint16_t SOFF

SOFF register, save offset bytes every transfer

__IO uint16_t ATTR

ATTR register, source/destination transfer size and modulo

__IO uint32_t NBYTES

Nbytes register, minor loop length in bytes

__IO uint32_t SLAST

SLAST register

__IO uint32_t DADDR

DADDR register, used for destination address

__IO uint16_t DOFF

DOFF register, used for destination offset

__IO uint16_t CITER

CITER register, current minor loop numbers, for unfinished minor loop.

__IO uint32_t DLAST_SGA

DLASTSGA register, next stcd address used in scatter-gather mode

__IO uint16_t CSR

CSR register, for TCD control status

__IO uint16_t BITER

BITER register, begin minor loop count.

struct _edma_handle

#include <fsl_edma.h> eDMA transfer handle structure

Public Members

edma_callback callback

Callback function for major count exhausted.

void *userData

Callback function parameter.

DMA_Type *base

eDMA peripheral base address.

`edma_tcd_t *tcdPool`
Pointer to memory stored TCDs.

`uint8_t channel`
eDMA channel number.

`volatile int8_t header`
The first TCD index.

`volatile int8_t tail`
The last TCD index.

`volatile int8_t tcdUsed`
The number of used TCD slots.

`volatile int8_t tcdSize`
The total number of TCD slots in the queue.

`uint8_t flags`
The status of the current channel.

2.13 ELEMU: Edgelock Messaging unit driver

Values:

enumerator `kStatus_ELEMU_AgumentOutOfRange`
ELEMU status for out of range access.

enumerator `kStatus_ELEMU_InvalidArgument`
ELEMU status for invalid argument check.

enumerator `kStatus_ELEMU_RequestTimeout`
ELEMU status for timeout.

enumerator `kStatus_ELEMU_Busy`
ELEMU status for reservation by other core.

enum `elemu_ownership_status_t`

Values:

enumerator `kStatus_ELEMU_Unknown`
Unexpected ELEMU ownership.

enumerator `kStatus_ELEMU_LockedByMe`
ELEMU reserved for current core.

enumerator `kStatus_ELEMU_LockedByOther`
ELEMU reserved for other core.

enumerator `kStatus_ELEMU_Free`
ELEMU not reserved.

typedef struct *mu_hdr* `mu_hdr_t`

void `ELEMU_mu_hal_send_data(ELEMU_Type *mu, uint8_t regid, uint32_t *data)`

Send one word of data to specified transmit register.

This function waits until the transmit register is ready and writes one word of data.

Parameters

- `mu` – ELEMU peripheral base address
- `regid` – Transmit register index
- `data` – Pointer to data to be sent

`void ELEMU_mu_hal_receive_data(ELEMU_Type *mu, uint8_t regid, uint32_t *data)`

Receive one word of data from specified receive register.

This function waits until data is available in the receive register and reads it.

Parameters

- `mu` – ELEMU peripheral base address
- `regid` – Receive register index
- `data` – Pointer to store received data

`status_t ELEMU_mu_hal_receive_data_wait(ELEMU_Type *mu, uint8_t regid, uint32_t *data, uint32_t wait)`

Wait for data availability and receive one word from specified register.

Parameters

- `mu` – ELEMU peripheral base address
- `regid` – Receive register index
- `data` – Pointer to store received data
- `wait` – Maximum wait iterations

Returns

`kStatus_Success` if successful, `kStatus_ELEMU_RequestTimeout` if timeout

`status_t ELEMU_mu_read_message(ELEMU_Type *mu, uint32_t *buf, uint8_t *size, uint8_t read_header)`

Read a message from ELEMU receive registers.

This function reads a message from ELEMU including header and payload.

Parameters

- `mu` – ELEMU peripheral base address
- `buf` – Pointer to buffer to store message
- `size` – Pointer to size variable
- `read_header` – Flag to indicate if header should be read

Returns

`kStatus_Success` if successful, `kStatus_ELEMU_InvalidArgument` if `buf` or `size` is `NULL`

`status_t ELEMU_mu_read_data_wait(ELEMU_Type *mu, uint32_t buf[], uint8_t *size, uint32_t wait)`

Wait and read data from ELEMU receive registers.

This function waits for data availability and reads the specified number of words.

Parameters

- `mu` – ELEMU peripheral base address
- `buf` – Pointer to buffer to store data
- `size` – Pointer to size variable
- `wait` – Maximum wait iterations

Returns

kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf or size is NULL, kStatus_ELEMU_RequestTimeout if timeout

status_t ELEMU_mu_send_message(ELEMU_Type *mu, uint32_t buf[], size_t wordCount)

Send a message to ELEMU using transmit registers.

This function sends a message buffer to ELEMU by writing words to transmit registers.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to message buffer
- wordCount – Number of words to send

Returns

kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_get_response(ELEMU_Type *mu, uint32_t *buf, size_t wordCount)

Get response message from ELEMU.

This function reads a response message from ELEMU using the read_message function.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to buffer to store response
- wordCount – Expected number of words

Returns

kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_wait_for_data(ELEMU_Type *mu, uint32_t *buf, size_t wordCount, uint32_t wait)

Wait for data availability and read from ELEMU.

This function waits for data and reads the specified number of words.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to buffer to store data
- wordCount – Expected number of words
- wait – Maximum wait iterations

Returns

kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_wait_for_ready(ELEMU_Type *mu, uint32_t wait)

Wait until ELEMU is ready for communication.

This function checks the status register and waits until ELEMU is ready.

Parameters

- mu – ELEMU peripheral base address
- wait – Maximum wait iterations

Returns

kStatus_Success if ready, kStatus_ELEMU_RequestTimeout if timeout

elemu_ownership_status_t ELEMU_mu_get_ownership_status(ELEMU_Type *mu)

Get current ownership status of ELEMU.

Parameters

- mu – ELEMU peripheral base address

Returns

Ownership status enum value

status_t ELEMU_mu_get_ownership(ELEMU_Type *mu)

Attempt to acquire ownership of ELEMU.

This function tries to acquire ownership of ELEMU using the semaphore register.

Parameters

- mu – ELEMU peripheral base address

Returns

kStatus_Success if acquired, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if acquisition failed

status_t ELEMU_mu_release_ownership(ELEMU_Type *mu)

Release ownership of ELEMU.

This function releases ownership of ELEMU using the semaphore release register.

Parameters

- mu – ELEMU peripheral base address

Returns

kStatus_Success if released, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if release failed

status_t ELEMU_mu_release_ownership_force(ELEMU_Type *mu)

Release ownership of ELEMU.

This function releases ownership of ELEMU using the semaphore release register.

Parameters

- mu – ELEMU peripheral base address

Returns

kStatus_Success if released, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if release failed

void ELEMU_mu_init(ELEMU_Type *mu)

Initialize ELEMU module.

This function does nothing. ELEMU is initialized after leaving ROM.

Parameters

- mu – ELEMU peripheral base address

status_t ELEMU_LP_WakeupPathInit(ELEMU_Type *mu)

Initialize low-power wakeup path for ELEMU.

This function enables clocks and waits for ELEMU startup sequence.

Parameters

- mu – ELEMU peripheral base address

Returns

kStatus_Success if successful, kStatus_Fail otherwise

```
struct mu_hdr
```

```
#include <fsl_lemu.h>
```

2.14 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

2.15 FGPIIO Driver

2.16 C40ESP3 Flash Driver

`enum _flash_driver_api_keys`
Enumeration for Flash driver API keys.

Note: The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Values:

`enumerator kFLASH_ApiEraseKey`
Key value used to validate all flash erase APIs.

`status_t FLASH_Init(flash_config_t *config)`
Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- `config` – Pointer to the storage for the driver runtime state.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.

`status_t FLASH_Erase(flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes, uint32_t key)`

Erases the flash sectors encompassed by parameters passed into function.

`status_t FLASH_EraseAll(FMU_Type *base, uint32_t key)`

Erases entire flash and ifr.

`status_t FLASH_Program(flash_config_t *config, FMU_Type *base, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`

Programs flash phrases with data at locations passed in through parameters.

`status_t FLASH_ProgramPage(flash_config_t *config, FMU_Type *base, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`

Programs flash pages with data at locations passed in through parameters.

status_t FLASH_VerifyErasePhrase(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash phrases are erased.

status_t FLASH_VerifyErasePage(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash pages are erased.

status_t FLASH_VerifyEraseSector(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash sectors are erased.

status_t FLASH_VerifyEraseAll(FMU_Type *base)

Verify that all flash and IFR space is erased.

status_t FLASH_VerifyEraseBlock(*flash_config_t* *config, FMU_Type *base, uint32_t blockaddr)

Verify that a flash block is erased.

status_t FLASH_VerifyEraseIFRPhrase(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr phrases are erased.

status_t FLASH_VerifyEraseIFRPage(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr pages are erased.

status_t FLASH_VerifyEraseIFRSector(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr sectors are erased.

status_t FLASH_GetProperty(*flash_config_t* *config, *flash_property_tag_t* whichProperty, uint32_t *value)

Returns the desired flash property.

status_t Read_Into_MISR(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t ending, uint32_t *seed, uint32_t *signature)

Read into MISR.

The Read into MISR operation generates a signature based on the contents of the selected flash memory using an embedded MISR.

status_t Read_IFR_Into_MISR(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t ending, uint32_t *seed, uint32_t *signature)

Read IFR into MISR.

The Read IFR into MISR operation generates a signature based on the contents of the selected IFR space using an embedded MISR.

FSL_FLASH_DRIVER_VERSION

Flash driver version for SDK.

Version 2.3.3.

enum _flash_driver_version_constants

Flash driver version for ROM.

Values:

enumerator kFLASH_DriverVersionName

Flash driver version name.

enumerator kFLASH_DriverVersionMajor

Major flash driver version.

enumerator kFLASH_DriverVersionMinor
Minor flash driver version.

enumerator kFLASH_DriverVersionBugfix
Bugfix for flash driver version.

enum _flash_property_tag
Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflash0SectorSize
Pflash sector size property.

enumerator kFLASH_PropertyPflash0TotalSize
Pflash total size property.

enumerator kFLASH_PropertyPflash0BlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflash0BlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflash0BlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyPflash0FacSupport
Pflash fac support property.

enumerator kFLASH_PropertyPflash0AccessSegmentSize
Pflash access segment size property.

enumerator kFLASH_PropertyPflash0AccessSegmentCount
Pflash access segment count property.

enumerator kFLASH_PropertyPflash1SectorSize
Pflash sector size property.

enumerator kFLASH_PropertyPflash1TotalSize
Pflash total size property.

enumerator kFLASH_PropertyPflash1BlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflash1BlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflash1BlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyPflash1FacSupport
Pflash fac support property.

enumerator kFLASH_PropertyPflash1AccessSegmentSize
Pflash access segment size property.

enumerator kFLASH_PropertyPflash1AccessSegmentCount
Pflash access segment count property.

enumerator kFLASH_PropertyFlexRamBlockBaseAddr
FlexRam block base address property.

enumerator kFLASH_PropertyFlexRamTotalSize

FlexRam total size property.

typedef enum *_flash_property_tag* flash__property__tag__t

Enumeration for various flash properties.

typedef struct *_flash_mem_descriptor* flash__mem__desc__t

Flash memory descriptor.

typedef struct *_flash_ifr_desc* flash__ifr__desc__t

typedef struct *_msf1_config* msf1__config__t

typedef struct *_flash_config* flash__config__t

Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

FLASH__ADDR__MASK

struct *_flash_mem_descriptor*

#include <fsl_k4_flash.h> Flash memory descriptor.

Public Members

uint32_t blockBase

Base address of the flash block

uint32_t totalSize

The size of the flash block.

uint32_t blockCount

A number of flash blocks.

struct *_flash_ifr_desc*

#include <fsl_k4_flash.h>

struct *_msf1_config*

#include <fsl_k4_flash.h>

struct *_flash_config*

#include <fsl_k4_flash.h> Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

2.17 FlexCAN: Flex Controller Area Network Driver

2.18 FlexCAN Driver

bool FLEXCAN_IsInstanceHasFDMode(CAN_Type *base)

Determine whether the FlexCAN instance support CAN FD mode at run time.

Note: Use this API only if different soc parts share the SOC part name macro define. Otherwise, a different SOC part name can be used to determine at compile time whether the FlexCAN instance supports CAN FD mode or not. If need use this API to determine if CAN FD

mode is supported, the FLEXCAN_Init function needs to be executed first, and then call this API and use the return to value determines whether to supports CAN FD mode, if return true, continue calling FLEXCAN_FDInit to enable CAN FD mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

return TRUE if instance support CAN FD mode, FALSE if instance only support classic CAN (2.0) mode.

uint32_t FLEXCAN_GetFDMailboxOffset(CAN_Type *base, uint8_t mbIdx)

Get Mailbox offset number by dword.

This function gets the offset number of the specified mailbox. Mailbox is not consecutive between memory regions when payload is not 8 bytes so need to calculate the specified mailbox address. For example, in the first memory region, MB[0].CS address is 0x4002_4080. For 32 bytes payload frame, the second mailbox is $((1/12)*512 + 1\%12*40)/4 = 10$, meaning 10 dword after the 0x4002_4080, which is actually the address of mailbox MB[1].CS.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – Mailbox index.

Returns

Mailbox address offset in word.

status_t FLEXCAN_EnterFreezeMode(CAN_Type *base)

Enter FlexCAN Freeze Mode.

This function makes the FlexCAN work under Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

status_t FLEXCAN_ExitFreezeMode(CAN_Type *base)

Exit FlexCAN Freeze Mode.

This function makes the FlexCAN leave Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

uint32_t FLEXCAN_GetInstance(CAN_Type *base)

Get the FlexCAN instance from peripheral base address.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN instance.


```
bool FLEXCAN_CalculateImprovedTimingValues(CAN_Type *base, uint32_t bitRate, uint32_t
                                         sourceClock_Hz, flexcan_timing_config_t
                                         *pTimingConfig)
```

Calculates the improved timing values by specific bit Rates for classical CAN.

This function use to calculates the Classical CAN timing values according to the given bit rate. The Calculated timing values will be set in CTRL1/CBT/ENCBT register. The calculation is based on the recommendation of the CiA 301 v4.2.0 and previous version document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The classical CAN speed in bps defined by user, should be less than or equal to 1Mbps.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration.

```
void FLEXCAN_Init(CAN_Type *base, const flexcan_config_t *pConfig, uint32_t sourceClock_Hz)
```

Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_Init function by passing in these parameters.

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc      = kFLEXCAN_ClkSrc0;
flexcanConfig.bitRate     = 1000000U;
flexcanConfig.maxMbNum    = 16;
flexcanConfig.enableLoopBack = false;
flexcanConfig.enableSelfWakeup = false;
flexcanConfig.enableIndividMask = false;
flexcanConfig.enableDoze     = false;
flexcanConfig.disableSelfReception = false;
flexcanConfig.enableListenOnlyMode = false;
flexcanConfig.timingConfig   = timingConfig;
FLEXCAN_Init(CAN0, &flexcanConfig, 4000000UL);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.

```
bool FLEXCAN_FDCalculateImprovedTimingValues(CAN_Type *base, uint32_t bitRate, uint32_t
                                             bitRateFD, uint32_t sourceClock_Hz,
                                             flexcan_timing_config_t *pTimingConfig)
```

Calculates the improved timing values by specific bit rates for CANFD.

This function use to calculates the CANFD timing values according to the given nominal phase bit rate and data phase bit rate. The Calculated timing values will be set in CBT/ENCBT and FDCBT/EDCBT registers. The calculation is based on the recommendation of the CiA 1301 v1.0.0 document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The CANFD bus control speed in bps defined by user.

- bitRateFD – The CAN FD data phase speed in bps defined by user. Equal to bitRate means disable bit rate switching.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

```
void FLEXCAN_FDInit(CAN_Type *base, const flexcan_config_t *pConfig, uint32_t  
sourceClock_Hz, flexcan_mb_size_t dataSize, bool brs)
```

Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_FDInit function by passing in these parameters.

```
flexcan_config_t flexcanConfig;  
flexcanConfig.clkSrc      = kFLEXCAN_ClkSrc0;  
flexcanConfig.bitRate     = 1000000U;  
flexcanConfig.bitRateFD   = 2000000U;  
flexcanConfig.maxMbNum    = 16;  
flexcanConfig.enableLoopBack = false;  
flexcanConfig.enableSelfWakeup = false;  
flexcanConfig.enableIndividMask = false;  
flexcanConfig.disableSelfReception = false;  
flexcanConfig.enableListenOnlyMode = false;  
flexcanConfig.enableDoze    = false;  
flexcanConfig.timingConfig  = timingConfig;  
FLEXCAN_FDInit(CAN0, &flexcanConfig, 8000000UL, kFLEXCAN_16BperMB, true);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.
- dataSize – FlexCAN Message Buffer payload size. The actual transmitted or received CAN FD frame data size needs to be less than or equal to this value.
- brs – True if bit rate switch is enabled in FD mode.

```
void FLEXCAN_Deinit(CAN_Type *base)
```

De-initializes a FlexCAN instance.

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

- base – FlexCAN peripheral base address.

```
void FLEXCAN_GetDefaultConfig(flexcan_config_t *pConfig)
```

Gets the default configuration structure.

This function initializes the FlexCAN configuration structure to default values. The default values are as follows. flexcanConfig->clkSrc = kFLEXCAN_ClkSrc0; flexcanConfig->bitRate = 1000000U; flexcanConfig->bitRateFD = 2000000U; flexcanConfig->maxMbNum = 16; flexcanConfig->enableLoopBack = false; flexcanConfig->enableSelfWakeup = false; flexcanConfig->enableIndividMask = false; flexcanConfig->disableSelfReception = false; flexcanConfig->enableListenOnlyMode = false; flexcanConfig->enableDoze = false; flexcanConfig->

```
>enablePretendedeNetworking = false; flexcanConfig->enableMemoryErrorControl  
= true; flexcanConfig->enableNonCorrectableErrorEnterFreeze = true;  
flexcanConfig->enableTransceiverDelayMeasure = true; flexcanConfig->  
>enableRemoteRequestFrameStored = true; flexcanConfig->payloadEndianness = kFLEX-  
CAN_bigEndian; flexcanConfig.timingConfig = timingConfig;
```

Parameters

- pConfig – Pointer to the FlexCAN configuration structure.

void FLEXCAN_SetTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN classical CAN protocol timing characteristic.

This function gives user settings to classical CAN or CAN FD nominal phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetBitRate() instead.

Note: Calling FLEXCAN_SetTimingConfig() overrides the bit rate set in FLEXCAN_Init() or FLEXCAN_SetBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t bitRate_Bps)

Set bit rate of FlexCAN classical CAN frame or CAN FD frame nominal phase.

This function set the bit rate of classical CAN frame or CAN FD frame nominal phase base on FLEXCAN_CalculateImprovedTimingValues() API calculated timing values.

Note: Calling FLEXCAN_SetBitRate() overrides the bit rate set in FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRate_Bps – Bit rate in Bps.

Returns

kStatus_Success - Set CAN baud rate (only Nominal phase) successfully.

void FLEXCAN_SetFDTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN CANFD data phase timing characteristic.

This function gives user settings to CANFD data phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetFDBitRate() to set both Nominal/Data bit Rate instead.

Note: Calling FLEXCAN_SetFDTimingConfig() overrides the data phase bit rate set in FLEXCAN_FDInit()/FLEXCAN_SetFDBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

```
status_t FLEXCAN_SetFDBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t  
                             bitRateN_Bps, uint32_t bitRateD_Bps)
```

Set bit rate of FlexCAN FD frame.

This function set the baud rate of FLEXCAN FD base on FLEXCAN_FDCalculateImprovedTimingValues() API calculated timing values.

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRateN_Bps – Nominal bit Rate in Bps.
- bitRateD_Bps – Data bit Rate in Bps.

Returns

kStatus_Success - Set CAN FD bit rate (include Nominal and Data phase) successfully.

```
void FLEXCAN_SetRxMbGlobalMask(CAN_Type *base, uint32_t mask)
```

Sets the FlexCAN receive message buffer global mask.

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Message Buffer Global Mask value.

```
void FLEXCAN_SetRxFifoGlobalMask(CAN_Type *base, uint32_t mask)
```

Sets the FlexCAN receive FIFO global mask.

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Fifo Global Mask value.

```
void FLEXCAN_SetRxIndividualMask(CAN_Type *base, uint8_t maskIdx, uint32_t mask)
```

Sets the FlexCAN receive individual mask.

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the FLEXCAN_Init(). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

- base – FlexCAN peripheral base address.
- maskIdx – The Index of individual Mask.
- mask – Rx Individual Mask value.

```
void FLEXCAN_SetTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)
```

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

```
void FLEXCAN_SetRxMbConfig(CAN_Type *base, uint8_t mbIdx, const flexcan_rx_mb_config_t  
                           *pRxMbConfig, bool enable)
```

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer. User should invoke this API when CTRL2[RRS]=1. When CTRL2[RRS]=1, frame's ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbEmpty, kFLEXCAN_RxMbFull or kFLEXCAN_RxMbOverrun. Message buffer will store the remote frame in the same fashion of a data frame. No automatic remote response frame will be generated. User need to setup another message buffer to respond remote request.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

```
static inline void FLEXCAN_SetMbID(CAN_Type *base, uint8_t mbIdx, uint32_t id)
```

Configures a FlexCAN Message Buffer identifier.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- id – CAN Message Buffer Identifier, should use FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

```
void FLEXCAN_SetFDTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)
```

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

```
void FLEXCAN_SetFDRxMbConfig(CAN_Type *base, uint8_t mbIdx, const
                             flexcan_rx_mb_config_t *pRxMbConfig, bool enable)
```

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

```
static inline void FLEXCAN_SetFDMbID(CAN_Type *base, uint8_t mbIdx, uint32_t id)
```

Configures a FlexCAN Message Buffer identifier.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- id – CAN Message Buffer Identifier, should use FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

```
void FLEXCAN_SetRemoteResponseMbConfig(CAN_Type *base, uint8_t mbIdx, const
                                         flexcan_frame_t *pFrame)
```

Configures a FlexCAN Remote Response Message Buffer.

User should invoke this API when CTRL2[RRS]=0. When CTRL2[RRS]=0, frame's ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbRanswer. If there is a matching ID, then this mailbox content will be transmitted as response. The received remote request frame is not stored in receive buffer. It is only used to trigger a transmission of a frame in response.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pFrame – Pointer to CAN message frame structure for response.

```
void FLEXCAN_SetRxFifoConfig(CAN_Type *base, const flexcan_rx_fifo_config_t *pRxFifoConfig,
                             bool enable)
```

Configures the FlexCAN Legacy Rx FIFO.

This function configures the FlexCAN Rx FIFO with given configuration.

Note: Legacy Rx FIFO only can receive classic CAN message.

Parameters

- base – FlexCAN peripheral base address.
- pRxFifoConfig – Pointer to the FlexCAN Legacy Rx FIFO configuration structure. Can be NULL when enable parameter is false.

- enable – Enable/disable Legacy Rx FIFO.
 - true: Enable Legacy Rx FIFO.
 - false: Disable Legacy Rx FIFO.

```
void FLEXCAN_SetEnhancedRxFifoConfig(CAN_Type *base, const
                                     flexcan_enhanced_rx_fifo_config_t *pConfig, bool
                                     enable)
```

Configures the FlexCAN Enhanced Rx FIFO.

This function configures the Enhanced Rx FIFO with given configuration.

Note: Enhanced Rx FIFO support receive classic CAN or CAN FD messages, Legacy Rx FIFO and Enhanced Rx FIFO cannot be enabled at the same time.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Enhanced Rx FIFO configuration structure. Can be NULL when enable parameter is false.
- enable – Enable/disable Enhanced Rx FIFO.
 - true: Enable Enhanced Rx FIFO.
 - false: Disable Enhanced Rx FIFO.

```
void FLEXCAN_SetPNConfig(CAN_Type *base, const flexcan_pn_config_t *pConfig)
```

Configures the FlexCAN Pretended Networking mode.

This function configures the FlexCAN Pretended Networking mode with given configuration.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Rx FIFO configuration structure.

```
static inline uint64_t FLEXCAN_GetStatusFlags(CAN_Type *base)
```

Gets the FlexCAN module interrupt flags.

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators `_flexcan_flags`. To check the specific status, compare the return value with enumerators in `_flexcan_flags`.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the `_flexcan_flags`.

```
static inline void FLEXCAN_ClearStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears status flags with the provided mask.

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

- base – FlexCAN peripheral base address.
- mask – The status flags to be cleared, it is logical OR value of `_flexcan_flags`.

```
static inline void FLEXCAN_GetBusErrCount(CAN_Type *base, uint8_t *txErrBuf, uint8_t
                                         *rxErrBuf)
```

Gets the FlexCAN Bus Error Counter value.

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

- base – FlexCAN peripheral base address.
- txErrBuf – Buffer to store Tx Error Counter value.
- rxErrBuf – Buffer to store Rx Error Counter value.

```
static inline uint64_t FLEXCAN_GetMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Gets the FlexCAN low 64 Message Buffer interrupt flags.

This function gets the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

```
static inline uint64_t FLEXCAN_GetHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)
```

Gets the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

```
static inline void FLEXCAN_ClearMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears the FlexCAN low 64 Message Buffer interrupt flags.

This function clears the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_ClearHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
void FLEXCAN_GetMemoryErrorReportStatus(CAN_Type *base,
                                         flexcan_memory_error_report_status_t
                                         *errorStatus)
```

Gets the FlexCAN Memory Error Report registers status.

This function gets the FlexCAN Memory Error Report registers status.

Parameters

- base – FlexCAN peripheral base address.
- errorStatus – Pointer to FlexCAN Memory Error Report registers status structure.

static inline uint8_t FLEXCAN_GetPNMatchCount(CAN_Type *base)

Gets the FlexCAN Number of Matches when in Pretended Networking.

This function gets the number of times a given message has matched the predefined filtering criteria for ID and/or PL before a wakeup event.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of received wake up messages.

static inline uint32_t FLEXCAN_GetEnhancedFifoDataCount(CAN_Type *base)

Gets the number of FlexCAN Enhanced Rx FIFO available frames.

This function gets the number of CAN messages stored in the Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of available CAN messages stored in the Enhanced Rx FIFO.

static inline void FLEXCAN_EnableInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN interrupts according to the provided mask.

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.
- mask – The interrupts to enable. Logical OR of `_flexcan_interrupt_enable`.

static inline void FLEXCAN_DisableInterrupts(CAN_Type *base, uint64_t mask)

Disables FlexCAN interrupts according to the provided mask.

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.
- mask – The interrupts to disable. Logical OR of `_flexcan_interrupt_enable`.

static inline void FLEXCAN_EnableMbInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN low 64 Message Buffer interrupts.

This function enables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

static inline void FLEXCAN_EnableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_DisableMbInterrupts(CAN_Type *base, uint64_t mask)
```

Disables FlexCAN low 64 Message Buffer interrupts.

This function disables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_DisableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)
```

Disables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
void FLEXCAN_EnableRxFifoDMA(CAN_Type *base, bool enable)
```

Enables or disables the FlexCAN Rx FIFO DMA request.

This function enables or disables the DMA feature of FlexCAN build-in Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- enable – true to enable, false to disable.

```
static inline uintptr_t FLEXCAN_GetRxFifoHeadAddr(CAN_Type *base)
```

Gets the Rx FIFO Head address.

This function returns the FlexCAN Rx FIFO Head address, which is mainly used for the DMA/eDMA use case.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN Rx FIFO Head address.

```
static inline status_t FLEXCAN_Enable(CAN_Type *base, bool enable)
```

Enables or disables the FlexCAN module operation.

This function enables or disables the FlexCAN module.

Parameters

- base – FlexCAN base pointer.
- enable – true to enable, false to disable.

Returns

kStatus_Success Enable FlexCAN module successful
kStatus_Timeout Timeout when wait for Low-Power Mode Acknowledge

status_t FLEXCAN_WriteTxMb(CAN_Type *base, uint8_t mbIdx, const *flexcan_frame_t* *pTxFrame)

Writes a FlexCAN Message to the Transmit Message Buffer.

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadRxMb(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Receive Message Buffer.

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.

status_t FLEXCAN_WriteFDTxMb(CAN_Type *base, uint8_t mbIdx, const *flexcan_fd_frame_t* *pTxFrame)

Writes a FlexCAN FD Message to the Transmit Message Buffer.

This function writes a CAN FD Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN FD Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadFDRxMb(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN FD Message from Receive Message Buffer.

This function reads a CAN FD message from a specified Receive Message Buffer. The function fills a receive CAN FD message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_ReadRxFifo(CAN_Type *base, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Legacy Rx FIFO.

This function reads a CAN message from the FlexCAN Legacy Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.

status_t FLEXCAN_ReadEnhancedRxFifo(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN Message from Enhanced Rx FIFO.

This function reads a CAN or CAN FD message from the FlexCAN Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.

status_t FLEXCAN_ReadPNWakeUpMB(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Wake Up MB.

This function reads a CAN message from the FlexCAN Wake up Message Buffers. There are four Wake up Message Buffers (WMBs) used to store incoming messages in Pretended Networking mode. The WMB index indicates the arrival order. The last message is stored in WMB3.

Parameters

- base – FlexCAN peripheral base address.

- pRxFrame – Pointer to CAN message frame structure for reception.
- mbIdx – The FlexCAN Wake up Message Buffer index. Range in 0x0 ~ 0x3.

Return values

- kStatus__Success – - Read Message from Wake up Message Buffer successfully.
- kStatus__Fail – - Wake up Message Buffer has no valid content.

status_t FLEXCAN__TransferFDSendBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pTxFrame)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus__Success – - Write Tx Message Buffer Successfully.
- kStatus__Fail – - Tx Message Buffer is currently in use.
- kStatus__Timeout – - Failed to send frames within specific time.

status_t FLEXCAN__TransferFDReceiveBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN FD Message Buffer index.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus__Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN__RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus__Fail – - Rx Message Buffer is empty.
- kStatus__Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN__TransferFDSendNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN FD Message Buffer transfer structure. See the `flexcan_mb_transfer_t`.

Return values

- `kStatus_Success` – Start Tx Message Buffer sending process successfully.
- `kStatus_Fail` – Write Tx Message Buffer failed.
- `kStatus_FLEXCAN_TxBusy` – Tx Message Buffer is in use.

`status_t` FLEXCAN_TransferFDReceiveNonBlocking(*CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *pMbXfer*)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN FD Message Buffer transfer structure. See the `flexcan_mb_transfer_t`.

Return values

- `kStatus_Success` -- Start Rx Message Buffer receiving process successfully.
- `kStatus_FLEXCAN_RxBusy` -- Rx Message Buffer is in use.

`void` FLEXCAN_TransferFDAbortSend(*CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx*)

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN FD Message Buffer index.

`void` FLEXCAN_TransferFDAbortReceive(*CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx*)

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN FD Message Buffer index.

`status_t` FLEXCAN_TransferSendBlocking(*CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *pTxFrame*)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.
- kStatus_Timeout – - Failed to send frames within specific time.

status_t FLEXCAN_TransferReceiveBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveFifoBlocking(CAN_Type *base, *flexcan_frame_t* *pRxFrame)

Performs a polling receive transaction from Legacy Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveEnhancedFifoBlocking(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction from Enhanced Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.
- kStatus_Timeout – Failed to receive frames within specific time.

```
void FLEXCAN_TransferCreateHandle(CAN_Type *base, flexcan_handle_t *handle,  
                                flexcan_transfer_callback_t callback, void *userData)
```

Initializes the FlexCAN handle.

This function initializes the FlexCAN handle, which can be used for other FlexCAN transactional APIs. Usually, for a specified FlexCAN instance, call this API once to get the initialized handle.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

```
status_t FLEXCAN_TransferSendNonBlocking(CAN_Type *base, flexcan_handle_t *handle,  
                                         flexcan_mb_transfer_t *pMbXfer)
```

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

```
status_t FLEXCAN_TransferReceiveNonBlocking(CAN_Type *base, flexcan_handle_t *handle,  
                                             flexcan_mb_transfer_t *pMbXfer)
```

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- `kStatus_Success` -- Start Rx Message Buffer receiving process successfully.
- `kStatus_FLEXCAN_RxBusy` -- Rx Message Buffer is in use.

`status_t FLEXCAN_TransferReceiveFifoNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_fifo_transfer_t *pFifoXfer)`

Receives a message from Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `pFifoXfer` – FlexCAN Rx FIFO transfer structure. See the `flexcan_fifo_transfer_t`.

Return values

- `kStatus_Success` -- Start Rx FIFO receiving process successfully.
- `kStatus_FLEXCAN_RxFifoBusy` -- Rx FIFO is currently in use.

`status_t FLEXCAN_TransferGetReceiveFifoCount(CAN_Type *base, flexcan_handle_t *handle, size_t *count)`

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `count` – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`status_t FLEXCAN_TransferReceiveEnhancedFifoNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_fifo_transfer_t *pFifoXfer)`

Receives a message from Enhanced Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `pFifoXfer` – FlexCAN Rx FIFO transfer structure. See the ref `flexcan_fifo_transfer_t`.

Return values

- `kStatus_Success` -- Start Rx FIFO receiving process successfully.
- `kStatus_FLEXCAN_RxFifoBusy` -- Rx FIFO is currently in use.

```
static inline status_t FLEXCAN_TransferGetReceiveEnhancedFifoCount(CAN_Type *base,  
                                                                    flexcan_handle_t *handle,  
                                                                    size_t *count)
```

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- *kStatus_InvalidArgument* – count is Invalid.
- *kStatus_Success* – Successfully return the count.

```
uint32_t FLEXCAN_GetTimeStamp(flexcan_handle_t *handle, uint8_t mbIdx)
```

Gets the detail index of Mailbox's Timestamp by handle.

Then function can only be used when calling non-blocking Data transfer (TX/RX) API, After TX/RX data transfer done (User can get the status by handler's callback function), we can get the detail index of Mailbox's timestamp by handle, Detail non-blocking data transfer API (TX/RX) contain. -FLEXCAN_TransferSendNonBlocking -FLEXCAN_TransferFDSendNonBlocking -FLEXCAN_TransferReceiveNonBlocking -FLEXCAN_TransferFDReceiveNonBlocking -FLEXCAN_TransferReceiveFifoNonBlocking

Parameters

- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

Return values

the – index of mailbox 's timestamp stored in the handle.

```
static inline uint32_t FLEXCAN_GetHighResolutionTimeStamp(CAN_Type *base, uint8_t mbIdx)
```

```
void FLEXCAN_TransferAbortSend(CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)
```

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

```
void FLEXCAN_TransferAbortReceive(CAN_Type *base, flexcan_handle_t *handle, uint8_t  
                                  mbIdx)
```

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

`void FLEXCAN_TransferAbortReceiveFifo(CAN_Type *base, flexcan_handle_t *handle)`

Aborts the interrupt driven message receive from Rx FIFO process.

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

`void FLEXCAN_TransferAbortReceiveEnhancedFifo(CAN_Type *base, flexcan_handle_t *handle)`

Aborts the interrupt driven message receive from Enhanced Rx FIFO process.

This function aborts the interrupt driven message receive from Enhanced Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

`void FLEXCAN_TransferHandleIRQ(CAN_Type *base, flexcan_handle_t *handle)`

FlexCAN IRQ handle function.

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

`void FLEXCAN_MbHandleIRQ(CAN_Type *base, flexcan_handle_t *handle, uint32_t startMbIdx, uint32_t endMbIdx)`

FlexCAN Message Buffer IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- startMbIdx – First Message Buffer to handle.
- endMbIdx – Last Message Buffer to handle.

`void FLEXCAN_EhancedRxFifoHandleIRQ(CAN_Type *base, flexcan_handle_t *handle)`

FlexCAN Enhanced Rx FIFO IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

`void FLEXCAN_BusoffErrorHandleIRQ(CAN_Type *base, flexcan_handle_t *handle)`

FlexCAN Bus Off, Error and Warning IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

`void FLEXCAN_PNWakeUpHandleIRQ(CAN_Type *base, flexcan_handle_t *handle)`

FlexCAN Pretended Networking Wake-up IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.

- handle – FlexCAN handle pointer.

void FLEXCAN_MemoryErrorHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN Memory Error IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

FSL_FLEXCAN_DRIVER_VERSION

FlexCAN driver version.

FlexCAN transfer status.

Values:

enumerator kStatus_FLEXCAN_TxBusy

Tx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_TxIdle

Tx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_TxSwitchToRx

Remote Message is send out and Message buffer changed to Receive one.

enumerator kStatus_FLEXCAN_RxBusy

Rx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_RxIdle

Rx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_RxOverflow

Rx Message Buffer is Overflowed.

enumerator kStatus_FLEXCAN_RxFifoBusy

Rx Message FIFO is Busy.

enumerator kStatus_FLEXCAN_RxFifoIdle

Rx Message FIFO is Idle.

enumerator kStatus_FLEXCAN_RxFifoOverflow

Rx Message FIFO is overflowed.

enumerator kStatus_FLEXCAN_RxFifoWarning

Rx Message FIFO is almost overflowed.

enumerator kStatus_FLEXCAN_RxFifoDisabled

Rx Message FIFO is disabled during reading.

enumerator kStatus_FLEXCAN_ErrorStatus

FlexCAN Module Error and Status.

enumerator kStatus_FLEXCAN_WakeUp

FlexCAN is waken up from STOP mode.

enumerator kStatus_FLEXCAN_UnHandled

UnHadled Interrupt asserted.

enumerator kStatus_FLEXCAN_RxRemote

Rx Remote Message Received in Mail box.

enumerator kStatus_FLEXCAN_RxFifoUnderflow
Enhanced Rx Message FIFO is underflow.

enumerator kStatus_FLEXCAN_MemoryError
FlexCAN Memory Error.

enum _flexcan_frame_format
FlexCAN frame format.

Values:

enumerator kFLEXCAN_FrameFormatStandard
Standard frame format attribute.

enumerator kFLEXCAN_FrameFormatExtend
Extend frame format attribute.

enum _flexcan_frame_type
FlexCAN frame type.

Values:

enumerator kFLEXCAN_FrameTypeData
Data frame type attribute.

enumerator kFLEXCAN_FrameTypeRemote
Remote frame type attribute.

enum _flexcan_clock_source
FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

Values:

enumerator kFLEXCAN_ClkSrcOsc
FlexCAN Protocol Engine clock from Oscillator.

enumerator kFLEXCAN_ClkSrcPeri
FlexCAN Protocol Engine clock from Peripheral Clock.

enumerator kFLEXCAN_ClkSrc0
FlexCAN Protocol Engine clock selected by user as SRC == 0.

enumerator kFLEXCAN_ClkSrc1
FlexCAN Protocol Engine clock selected by user as SRC == 1.

enum _flexcan_wake_up_source
FlexCAN wake up source.

Values:

enumerator kFLEXCAN_WakeupSrcUnfiltered
FlexCAN uses unfiltered Rx input to detect edge.

enumerator kFLEXCAN_WakeupSrcFiltered
FlexCAN uses filtered Rx input to detect edge.

enum `_flexcan_endianness`

FlexCAN payload endianness.

Values:

enumerator `kFLEXCAN_bigEndian`

Transmit frame with MSB first, receive frame with big-endian format.

enumerator `kFLEXCAN_littleEndian`

Transmit frame with LSB first, receive frame with little-endian format.

enum `_flexcan_MB_timestamp_base`

FlexCAN timebase used for capturing 16-bit `TIME_STAMP` field of message buffer.

Values:

enumerator `kFLEXCAN_CANTimer`

FlexCAN free-running timer.

enumerator `kFLEXCAN_Lower16bitsHRTimer`

Lower 16 bits of high-resolution on-chip timer.

enumerator `kFLEXCAN_Upper16bitsHRTimer`

Upper 16 bits of high-resolution on-chip timer.

enum `_flexcan_capture_point`

FlexCAN capture point of 32-bit high resolution timebase during a CAN frame.

Values:

enumerator `kFLEXCAN_CANFrameID2ndBit`

Second bit of identifier field of any frame is on the CAN bus. `HR_TIME_STAMPn` register will not capture 32-bit counter value.

enumerator `kFLEXCAN_CANFrameEnd`

End of the CAN frame.

enumerator `kFLEXCAN_CANFrameStart`

Start of the CAN frame.

enumerator `kFLEXCAN_CANFDFrameRes`

Start of frame for classical CAN frames; res bit for CAN FD frames.

enum `_flexcan_rx_fifo_filter_type`

FlexCAN Rx Fifo Filter type.

Values:

enumerator `kFLEXCAN_RxFifoFilterTypeA`

One full ID (standard and extended) per ID Filter element.

enumerator `kFLEXCAN_RxFifoFilterTypeB`

Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

enumerator `kFLEXCAN_RxFifoFilterTypeC`

Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

enumerator `kFLEXCAN_RxFifoFilterTypeD`

All frames rejected.

enum `_flexcan_mb_size`

FlexCAN Message Buffer Payload size.

Values:

enumerator kFLEXCAN_8BperMB

Selects 8 bytes per Message Buffer.

enumerator kFLEXCAN_16BperMB

Selects 16 bytes per Message Buffer.

enumerator kFLEXCAN_32BperMB

Selects 32 bytes per Message Buffer.

enumerator kFLEXCAN_64BperMB

Selects 64 bytes per Message Buffer.

enum _flexcan_fd_frame_length

FlexCAN CAN FD frame supporting data length (available DLC values).

For Tx, when the Data size corresponding to DLC value stored in the MB selected for transmission is larger than the MB Payload size, FlexCAN adds the necessary number of bytes with constant 0xCC pattern to complete the expected DLC. For Rx, when the Data size corresponding to DLC value received from the CAN bus is larger than the MB Payload size, the high order bytes that do not fit the Payload size will lose.

Values:

enumerator kFLEXCAN_0BperFrame

Frame contains 0 valid data bytes.

enumerator kFLEXCAN_1BperFrame

Frame contains 1 valid data bytes.

enumerator kFLEXCAN_2BperFrame

Frame contains 2 valid data bytes.

enumerator kFLEXCAN_3BperFrame

Frame contains 3 valid data bytes.

enumerator kFLEXCAN_4BperFrame

Frame contains 4 valid data bytes.

enumerator kFLEXCAN_5BperFrame

Frame contains 5 valid data bytes.

enumerator kFLEXCAN_6BperFrame

Frame contains 6 valid data bytes.

enumerator kFLEXCAN_7BperFrame

Frame contains 7 valid data bytes.

enumerator kFLEXCAN_8BperFrame

Frame contains 8 valid data bytes.

enumerator kFLEXCAN_12BperFrame

Frame contains 12 valid data bytes.

enumerator kFLEXCAN_16BperFrame

Frame contains 16 valid data bytes.

enumerator kFLEXCAN_20BperFrame

Frame contains 20 valid data bytes.

enumerator kFLEXCAN_24BperFrame

Frame contains 24 valid data bytes.

enumerator kFLEXCAN_32BperFrame
Frame contains 32 valid data bytes.

enumerator kFLEXCAN_48BperFrame
Frame contains 48 valid data bytes.

enumerator kFLEXCAN_64BperFrame
Frame contains 64 valid data bytes.

enum _flexcan_efifo_dma_per_read_length
FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

Values:

enumerator kFLEXCAN_1WordPerRead
Transfer 1 32-bit words (CS).

enumerator kFLEXCAN_2WordPerRead
Transfer 2 32-bit words (CS + ID).

enumerator kFLEXCAN_3WordPerRead
Transfer 3 32-bit words (CS + ID + 1~4 bytes data).

enumerator kFLEXCAN_4WordPerRead
Transfer 4 32-bit words (CS + ID + 5~8 bytes data).

enumerator kFLEXCAN_5WordPerRead
Transfer 5 32-bit words (CS + ID + 9~12 bytes data).

enumerator kFLEXCAN_6WordPerRead
Transfer 6 32-bit words (CS + ID + 13~16 bytes data).

enumerator kFLEXCAN_7WordPerRead
Transfer 7 32-bit words (CS + ID + 17~20 bytes data).

enumerator kFLEXCAN_8WordPerRead
Transfer 8 32-bit words (CS + ID + 21~24 bytes data).

enumerator kFLEXCAN_9WordPerRead
Transfer 9 32-bit words (CS + ID + 25~28 bytes data).

enumerator kFLEXCAN_10WordPerRead
Transfer 10 32-bit words (CS + ID + 29~32 bytes data).

enumerator kFLEXCAN_11WordPerRead
Transfer 11 32-bit words (CS + ID + 33~36 bytes data).

enumerator kFLEXCAN_12WordPerRead
Transfer 12 32-bit words (CS + ID + 37~40 bytes data).

enumerator kFLEXCAN_13WordPerRead
Transfer 13 32-bit words (CS + ID + 41~44 bytes data).

enumerator kFLEXCAN_14WordPerRead
Transfer 14 32-bit words (CS + ID + 45~48 bytes data).

enumerator kFLEXCAN_15WordPerRead
Transfer 15 32-bit words (CS + ID + 49~52 bytes data).

enumerator kFLEXCAN_16WordPerRead
Transfer 16 32-bit words (CS + ID + 53~56 bytes data).

enumerator kFLEXCAN_17WordPerRead

Transfer 17 32-bit words (CS + ID + 57~60 bytes data).

enumerator kFLEXCAN_18WordPerRead

Transfer 18 32-bit words (CS + ID + 61~64 bytes data).

enumerator kFLEXCAN_19WordPerRead

Transfer 19 32-bit words (CS + ID + 64 bytes data + ID HIT).

enumerator kFLEXCAN_20WordPerRead

Transfer 20 32-bit words (CS + ID + 64 bytes data + ID HIT + HR timestamp).

enum _flexcan_rx_fifo_priority

FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

Values:

enumerator kFLEXCAN_RxFifoPrioLow

Matching process start from Rx Message Buffer first.

enumerator kFLEXCAN_RxFifoPrioHigh

Matching process start from Enhanced/Legacy Rx FIFO first.

enum _flexcan_interrupt_enable

FlexCAN interrupt enable enumerations.

This provides constants for the FlexCAN interrupt enable enumerations for use in the FlexCAN functions.

Note: FlexCAN Message Buffers and Legacy Rx FIFO interrupts not included in.

Values:

enumerator kFLEXCAN_BusOffInterruptEnable

Bus Off interrupt, use bit 15.

enumerator kFLEXCAN_ErrorInterruptEnable

CAN Error interrupt, use bit 14.

enumerator kFLEXCAN_TxWarningInterruptEnable

Tx Warning interrupt, use bit 11.

enumerator kFLEXCAN_RxWarningInterruptEnable

Rx Warning interrupt, use bit 10.

enumerator kFLEXCAN_FDErrorInterruptEnable

CAN FD Error interrupt, use bit 31.

enumerator kFLEXCAN_PNMatchWakeUpInterruptEnable

PN Match Wake Up interrupt, use high word bit 17.

enumerator kFLEXCAN_PNTimeoutWakeUpInterruptEnable

PN Timeout Wake Up interrupt, use high word bit 16. Enhanced Rx FIFO Underflow interrupt, use high word bit 31.

enumerator kFLEXCAN_ERxFifoUnderflowInterruptEnable

Enhanced Rx FIFO Overflow interrupt, use high word bit 30.

enumerator kFLEXCAN_ERxFifoOverflowInterruptEnable

Enhanced Rx FIFO Watermark interrupt, use high word bit 29.

enumerator kFLEXCAN_ERxFifoWatermarkInterruptEnable

Enhanced Rx FIFO Data Available interrupt, use high word bit 28.

enumerator kFLEXCAN_ERxFifoDataAvlInterruptEnable

enumerator kFLEXCAN_HostAccessNCErrInterruptEnable

Host Access With Non-Correctable Errors interrupt, use high word bit 0.

enumerator kFLEXCAN_FlexCanAccessNCErrInterruptEnable

FlexCAN Access With Non-Correctable Errors interrupt, use high word bit 2.

enumerator kFLEXCAN_HostOrFlexCanCErrInterruptEnable

Host or FlexCAN Access With Correctable Errors interrupt, use high word bit 3.

enum _flexcan_flags

FlexCAN status flags.

This provides constants for the FlexCAN status flags for use in the FlexCAN functions.

Note: The CPU read action clears the bits corresponding to the FLEXCAN_ErrorFlag macro, therefore user need to read status flags and distinguish which error is occur using _flexcan_error_flags enumerations.

Values:

enumerator kFLEXCAN_ErrorOverrunFlag

Error Overrun Status.

enumerator kFLEXCAN_FDErrIntFlag

CAN FD Error Interrupt Flag.

enumerator kFLEXCAN_BusoffDoneIntFlag

Bus Off process completed Interrupt Flag.

enumerator kFLEXCAN_SynchFlag

CAN Synchronization Status.

enumerator kFLEXCAN_TxWarningIntFlag

Tx Warning Interrupt Flag.

enumerator kFLEXCAN_RxWarningIntFlag

Rx Warning Interrupt Flag.

enumerator kFLEXCAN_IdleFlag

FlexCAN In IDLE Status.

enumerator kFLEXCAN_FaultConfinementFlag

FlexCAN Fault Confinement State.

enumerator kFLEXCAN_TransmittingFlag

FlexCAN In Transmission Status.

enumerator kFLEXCAN_ReceivingFlag

FlexCAN In Reception Status.

enumerator kFLEXCAN_BusOffIntFlag

Bus Off Interrupt Flag.

enumerator kFLEXCAN_ErrorIntFlag
CAN Error Interrupt Flag.

enumerator kFLEXCAN_ErrorFlag

enumerator kFLEXCAN_PNMatchIntFlag
PN Matching Event Interrupt Flag.

enumerator kFLEXCAN_PNTimeoutIntFlag
PN Timeout Event Interrupt Flag.

enumerator kFLEXCAN_ERxFifoUnderflowIntFlag
Enhanced Rx FIFO underflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoOverflowIntFlag
Enhanced Rx FIFO overflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoWatermarkIntFlag
Enhanced Rx FIFO watermark Interrupt Flag.

enumerator kFLEXCAN_ERxFifoDataAvlIntFlag
Enhanced Rx FIFO data available Interrupt Flag.

enumerator kFLEXCAN_ERxFifoEmptyFlag
Enhanced Rx FIFO empty status.

enumerator kFLEXCAN_ERxFifoFullFlag
Enhanced Rx FIFO full status.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorIntFlag
Host Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorIntFlag
FlexCAN Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_CorrectableErrorIntFlag
Correctable Error Interrupt Flag.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorOverrunFlag
Host Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorOverrunFlag
FlexCAN Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_CorrectableErrorOverrunFlag
Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_AllMemoryErrorIntFlag
All Memory Error Interrupt Flags.

enumerator kFLEXCAN_AllMemoryErrorFlag
All Memory Error Flags.

enum _flexcan_error_flags
FlexCAN error status flags.

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFlexCAN_ErrorFlag in _flexcan_flags enumerations to determine which error is generated.

Values:

enumerator kFLEXCAN_FDStuffingError
Stuffing Error.

enumerator kFLEXCAN_FDFormError
Form Error.

enumerator kFLEXCAN_FDCrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_FDBit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_FDBit1Error
Unable to send recessive bit.

enumerator kFLEXCAN_TxErrorWarningFlag
Tx Error Warning Status.

enumerator kFLEXCAN_RxErrorWarningFlag
Rx Error Warning Status.

enumerator kFLEXCAN_StuffingError
Stuffing Error.

enumerator kFLEXCAN_FormError
Form Error.

enumerator kFLEXCAN_CrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_AckError
Received no ACK on transmission.

enumerator kFLEXCAN_Bit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_Bit1Error
Unable to send recessive bit.

FlexCAN Legacy Rx FIFO status flags.

The FlexCAN Legacy Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occuppies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Values:

enumerator kFLEXCAN_RxFifoOverflowFlag
Rx FIFO overflow flag.

enumerator kFLEXCAN_RxFifoWarningFlag
Rx FIFO almost full flag.

enumerator kFLEXCAN_RxFifoFrameAvlFlag
Frames available in Rx FIFO flag.

enum _flexcan_memory_error_type
FlexCAN Memory Error Type.

Values:

enumerator kFLEXCAN_CorrectableError
The memory error is correctable which means on bit error.

enumerator kFLEXCAN_NonCorrectableError

The memory error is non-correctable which means two bit errors.

enum _flexcan_memory_access_type

FlexCAN Memory Access Type.

Values:

enumerator kFLEXCAN_MoveOutFlexCanAccess

The memory error was detected during move-out FlexCAN access.

enumerator kFLEXCAN_MoveInAccess

The memory error was detected during move-in FlexCAN access.

enumerator kFLEXCAN_TxArbitrationAccess

The memory error was detected during Tx Arbitration FlexCAN access.

enumerator kFLEXCAN_RxMatchingAccess

The memory error was detected during Rx Matching FlexCAN access.

enumerator kFLEXCAN_MoveOutHostAccess

The memory error was detected during Rx Matching Host (CPU) access.

enum _flexcan_byte_error_syndrome

FlexCAN Memory Error Byte Syndrome.

Values:

enumerator kFLEXCAN_NoError

No bit error in this byte.

enumerator kFLEXCAN_ParityBits0Error

Parity bit 0 error in this byte.

enumerator kFLEXCAN_ParityBits1Error

Parity bit 1 error in this byte.

enumerator kFLEXCAN_ParityBits2Error

Parity bit 2 error in this byte.

enumerator kFLEXCAN_ParityBits3Error

Parity bit 3 error in this byte.

enumerator kFLEXCAN_ParityBits4Error

Parity bit 4 error in this byte.

enumerator kFLEXCAN_DataBits0Error

Data bit 0 error in this byte.

enumerator kFLEXCAN_DataBits1Error

Data bit 1 error in this byte.

enumerator kFLEXCAN_DataBits2Error

Data bit 2 error in this byte.

enumerator kFLEXCAN_DataBits3Error

Data bit 3 error in this byte.

enumerator kFLEXCAN_DataBits4Error

Data bit 4 error in this byte.

enumerator kFLEXCAN_DataBits5Error

Data bit 5 error in this byte.

enumerator kFLEXCAN_DataBits6Error

Data bit 6 error in this byte.

enumerator kFLEXCAN_DataBits7Error

Data bit 7 error in this byte.

enumerator kFLEXCAN_AllZeroError

All-zeros non-correctable error in this byte.

enumerator kFLEXCAN_AllOneError

All-ones non-correctable error in this byte.

enumerator kFLEXCAN_NonCorrectableErrors

Non-correctable error in this byte.

enum _flexcan_pn_match_source

FlexCAN Pretended Networking match source selection.

Values:

enumerator kFLEXCAN_PNMatSrcID

Message match with ID filtering.

enumerator kFLEXCAN_PNMatSrcIDAndData

Message match with ID filtering and payload filtering.

enum _flexcan_pn_match_mode

FlexCAN Pretended Networking mode match type.

Values:

enumerator kFLEXCAN_PNMatModeEqual

Match upon ID/Payload contents against an exact target value.

enumerator kFLEXCAN_PNMatModeGreater

Match upon an ID/Payload value greater than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeSmaller

Match upon an ID/Payload value smaller than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeRange

Match upon an ID/Payload value inside a range, greater than or equal to a specified lower limit, and smaller than or equal to a specified upper limit

typedef enum _flexcan_frame_format flexcan_frame_format_t

FlexCAN frame format.

typedef enum _flexcan_frame_type flexcan_frame_type_t

FlexCAN frame type.

typedef enum _flexcan_clock_source flexcan_clock_source_t

FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

typedef enum _flexcan_wake_up_source flexcan_wake_up_source_t

FlexCAN wake up source.

typedef enum *_flexcan_endianness* flexcan_endianness_t

FlexCAN payload endianness.

typedef enum *_flexcan_MB_timestamp_base* flexcan_MB_timestamp_base_t

FlexCAN timebase used for capturing 16-bit TIME_STAMP field of message buffer.

typedef enum *_flexcan_capture_point* flexcan_capture_point_t

FlexCAN capture point of 32-bit high resolution timebase during a CAN frame.

typedef enum *_flexcan_rx_fifo_filter_type* flexcan_rx_fifo_filter_type_t

FlexCAN Rx Fifo Filter type.

typedef enum *_flexcan_mb_size* flexcan_mb_size_t

FlexCAN Message Buffer Payload size.

typedef enum *_flexcan_efifo_dma_per_read_length* flexcan_efifo_dma_per_read_length_t

FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

typedef enum *_flexcan_rx_fifo_priority* flexcan_rx_fifo_priority_t

FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

typedef enum *_flexcan_memory_error_type* flexcan_memory_error_type_t

FlexCAN Memory Error Type.

typedef enum *_flexcan_memory_access_type* flexcan_memory_access_type_t

FlexCAN Memory Access Type.

typedef enum *_flexcan_byte_error_syndrome* flexcan_byte_error_syndrome_t

FlexCAN Memory Error Byte Syndrome.

typedef struct *_flexcan_memory_error_report_status* flexcan_memory_error_report_status_t

FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of FLEXCAN_GetMemoryErrorReportStatus() function. And user can use FLEXCAN_GetMemoryErrorReportStatus to get the status of the last memory error access.

typedef struct *_flexcan_frame* flexcan_frame_t

FlexCAN message frame structure.

typedef struct *_flexcan_fd_frame* flexcan_fd_frame_t

CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see *_flexcan_fd_frame_length*.

typedef struct *_flexcan_timing_config* flexcan_timing_config_t

FlexCAN protocol timing characteristic configuration structure.

typedef struct *_flexcan_config* flexcan_config_t

FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

typedef struct *flexcan_rx_mb_config* flexcan_rx_mb_config_t

FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

typedef enum *flexcan_pn_match_source* flexcan_pn_match_source_t

FlexCAN Pretended Networking match source selection.

typedef enum *flexcan_pn_match_mode* flexcan_pn_match_mode_t

FlexCAN Pretended Networking mode match type.

typedef struct *flexcan_pn_config* flexcan_pn_config_t

FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of FLEXCAN_SetPNConfig() function. The FLEXCAN_SetPNConfig() function is used to configure FlexCAN Networking work mode.

typedef struct *flexcan_rx_fifo_config* flexcan_rx_fifo_config_t

FlexCAN Legacy Rx FIFO configuration structure.

typedef struct *flexcan_enhanced_rx_fifo_std_id_filter* flexcan_enhanced_rx_fifo_std_id_filter_t

FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

typedef struct *flexcan_enhanced_rx_fifo_ext_id_filter* flexcan_enhanced_rx_fifo_ext_id_filter_t

FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

typedef struct *flexcan_enhanced_rx_fifo_config* flexcan_enhanced_rx_fifo_config_t

FlexCAN Enhanced Rx FIFO configuration structure.

typedef struct *flexcan_mb_transfer* flexcan_mb_transfer_t

FlexCAN Message Buffer transfer.

typedef struct *flexcan_fifo_transfer* flexcan_fifo_transfer_t

FlexCAN Rx FIFO transfer.

typedef struct *flexcan_handle* flexcan_handle_t

FlexCAN handle structure definition.

typedef void (*flexcan_transfer_callback_t)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint64_t result, void *userData)

FLEXCAN_WAIT_TIMEOUT

FLEXCAN_POLLING_TIMEOUT

Max loops to wait for polling transfer.

FLEXCAN_MODULE_TIMEOUT

Max loops to wait for FlexCAN register access complete.

DLC_LENGTH_DECODE(dlc)

FlexCAN frame length helper macro.

FLEXCAN_ID_STD(id)

FlexCAN Frame ID helper macro.

Standard Frame ID helper macro.

FLEXCAN_ID_EXT(id)

Extend Frame ID helper macro.

`FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)`

FlexCAN Rx Message Buffer Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

`FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)`

Extend Rx Message Buffer Mask helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)`

FlexCAN Legacy Rx FIFO Mask helper macro.

Standard Rx FIFO Mask helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)`

Standard Rx FIFO Mask helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)`

Standard Rx FIFO Mask helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id)`

Standard Rx FIFO Mask helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id)`

Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id)`

Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)`

Standard Rx FIFO Mask helper macro Type C lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)`

Extend Rx FIFO Mask helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)`

Extend Rx FIFO Mask helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)`

Extend Rx FIFO Mask helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)`

Extend Rx FIFO Mask helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)`

Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)`

Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)`

Extend Rx FIFO Mask helper macro Type C lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide)`

FlexCAN Rx FIFO Filter helper macro.

Standard Rx FIFO Filter helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)`

Standard Rx FIFO Filter helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)`

Standard Rx FIFO Filter helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)`

Standard Rx FIFO Filter helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)`

Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)`

Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id)`

Standard Rx FIFO Filter helper macro Type C lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide)`

Extend Rx FIFO Filter helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)`

Extend Rx FIFO Filter helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)`

Extend Rx FIFO Filter helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id)`

Extend Rx FIFO Filter helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)`

Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)`

Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id)`

Extend Rx FIFO Filter helper macro Type C lower part helper macro.

`ENHANCED_RX_FIFO_FSCH(x)`

FlexCAN Enhanced Rx FIFO Filter and Mask helper macro.

`RTR_STD_HIGH(x)`

`RTR_STD_LOW(x)`

`RTR_EXT(x)`

`ID_STD_LOW(id)`

`ID_STD_HIGH(id)`

`ID_EXT(id)`

`FLEXCAN_ENHANCED_RX_FIFO_STD_MASK_AND_FILTER(id, rtr, id_mask, rtr_mask)`

Standard ID filter element with filter + mask scheme.

`FLEXCAN_ENHANCED_RX_FIFO_STD_FILTER_WITH_RANGE(id_upper, rtr, id_lower,
rtr_mask)`

Standard ID filter element with filter range.

`FLEXCAN_ENHANCED_RX_FIFO_STD_TWO_FILTERS(id1, rtr1, id2, rtr2)`

Standard ID filter element with two filters without masks.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_LOW(id, rtr)`

Extended ID filter element with filter + mask scheme low word.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_HIGH(id_mask, rtr_mask)`

Extended ID filter element with filter + mask scheme high word.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_LOW(id_upper, rtr)`

Extended ID filter element with range scheme low word.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_HIGH(id_lower, rtr_mask)`

Extended ID filter element with range scheme high word.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_LOW(id2, rtr2)`

Extended ID filter element with two filters without masks low word.

`FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_HIGH(id1, rtr1)`

Extended ID filter element with two filters without masks high word.

`FLEXCAN_PN_STD_MASK(id, rtr)`

FlexCAN Pretended Networking ID Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

`FLEXCAN_PN_EXT_MASK(id, rtr)`

Extend Rx Message Buffer Mask helper macro.

`FLEXCAN_PN_INT_MASK(x)`

FlexCAN interrupt/status flag helper macro.

`FLEXCAN_PN_INT_UNMASK(x)`

`FLEXCAN_PN_STATUS_MASK(x)`

`FLEXCAN_PN_STATUS_UNMASK(x)`

`FLEXCAN_EFIFO_INT_MASK(x)`

`FLEXCAN_EFIFO_INT_UNMASK(x)`

`FLEXCAN_EFIFO_STATUS_MASK(x)`

`FLEXCAN_EFIFO_STATUS_UNMASK(x)`

`FLEXCAN_MECR_INT_MASK(x)`

`FLEXCAN_MECR_INT_UNMASK(x)`

`FLEXCAN_MECR_STATUS_MASK(x)`

`FLEXCAN_MECR_STATUS_UNMASK(x)`

`FLEXCAN_ERROR_AND_STATUS_INT_FLAG`

`FLEXCAN_PNWAKE_UP_FLAG`

`FLEXCAN_WAKE_UP_FLAG`

`FLEXCAN_MEMORY_ERROR_INT_FLAG`

`FLEXCAN_ENHANCED_RX_FIFO_INT_FLAG`

FlexCAN Enhanced Rx FIFO base address helper macro.

`E_RX_FIFO(base)`

`FLEXCAN_CALLBACK(x)`

FlexCAN transfer callback function.

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to `kStatus_FLEXCAN_ErrorStatus`, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the

index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

struct `_flexcan_memory_error_report_status`

#include <fsl_flexcan.h> FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of FLEXCAN_GetMemoryErrorReportStatus() function. And user can use FLEXCAN_GetMemoryErrorReportStatus to get the status of the last memory error access.

Public Members

flexcan_memory_error_type_t errorType

The type of memory error that giving rise to the report.

flexcan_memory_access_type_t accessType

The type of memory access that giving rise to the memory error.

uint16_t accessAddress

The address where memory error detected.

uint32_t errorData

The raw data word read from memory with error.

struct `_flexcan_frame`

#include <fsl_flexcan.h> FlexCAN message frame structure.

struct `_flexcan_fd_frame`

#include <fsl_flexcan.h> CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see `_flexcan_fd_frame_length`.

Public Members

uint32_t idhit

Note: ID HIT offset is changed dynamically according to data length code (DLC), when DLC is 15, they will be located below. Using FLEXCAN_FixEnhancedRxFifoFrameIdHit API is recommended to ensure this idhit value is correct. CAN Enhanced Rx FIFO filter hit id (This value is only used in Enhanced Rx FIFO receive mode).

uint32_t hrtimestamp

Note: HR timestamp offset is changed dynamically according to data length code (DLC). External 32-bit on-chip timer high-resolution timestamp.

struct `_flexcan_timing_config`

#include <fsl_flexcan.h> FlexCAN protocol timing characteristic configuration structure.

Public Members

uint32_t preDivider

Classic CAN or CAN FD nominal phase bit rate prescaler.

uint32_t rJumpwidth

Classic CAN or CAN FD nominal phase Re-sync Jump Width.

uint32_t phaseSeg1

Classic CAN or CAN FD nominal phase Segment 1.

uint32_t phaseSeg2

Classic CAN or CAN FD nominal phase Segment 2.

uint32_t propSeg

Classic CAN or CAN FD nominal phase Propagation Segment.

uint32_t fpreDivider

CAN FD data phase bit rate prescaler.

uint32_t frJumpwidth

CAN FD data phase Re-sync Jump Width.

uint32_t fphaseSeg1

CAN FD data phase Phase Segment 1.

uint32_t fphaseSeg2

CAN FD data phase Phase Segment 2.

uint32_t fpropSeg

CAN FD data phase Propagation Segment.

struct flexcan_config

#include <fsl_flexcan.h> FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

Public Members

flexcan_clock_source_t clkSrc

Clock source for FlexCAN Protocol Engine.

flexcan_wake_up_source_t wakeupSrc

Wake up source selection.

uint8_t maxMbNum

The maximum number of Message Buffers used by user.

bool enableLoopBack

Enable or Disable Loop Back Self Test Mode.

bool enableTimerSync

Enable or Disable Timer Synchronization.

bool enableIndividMask

Enable or Disable Rx Individual Mask and Queue feature.

bool disableSelfReception

Enable or Disable Self Reflection.

`bool enableListenOnlyMode`

Enable or Disable Listen Only Mode.

`bool enableDoze`

Enable or Disable Doze Mode.

`bool enablePretendedeNetworking`

Enable or Disable the Pretended Networking mode.

`bool enableMemoryErrorControl`

Enable or Disable the memory errors detection and correction mechanism.

`bool enableNonCorrectableErrorEnterFreeze`

Enable or Disable Non-Correctable Errors In FlexCAN Access Put Device In Freeze Mode.

`bool enableTransceiverDelayMeasure`

Enable or Disable the transceiver delay measurement, when it is enabled, then the secondary sample point position is determined by the sum of the transceiver delay measurement plus the enhanced TDC offset.

`bool enableRemoteRequestFrameStored`

true: Store Remote Request Frame in the same fashion of data frame. false: Generate an automatic Remote Response Frame.

flexcan_endianness_t payloadEndianness

Selects the byte order for the payload of transmit and receive frames, see *flexcan_endianness_t*.

`bool enableExternalTimeTick`

true: External time tick clocks the free-running timer. false: FlexCAN bit clock clocks the free-running timer.

flexcan_MB_timestamp_base_t captureTimeBase

Timebase of message buffer 16-bit TIME_STAMP field.

flexcan_capture_point_t capturePoint

Point in time when 32-bit timebase is captured during CAN frame.

`struct _flexcan_rx_mb_config`

#include <fsl_flexcan.h> FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Public Members

`uint32_t id`

CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

flexcan_frame_format_t format

CAN Frame Identifier format(Standard of Extend).

flexcan_frame_type_t type

CAN Frame Type(Data or Remote for classical CAN only).

struct `_flexcan_pn_config`

#include <fsl_flexcan.h> FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of `FLEXCAN_SetPNConfig()` function. The `FLEXCAN_SetPNConfig()` function is used to configure FlexCAN Networking work mode.

Public Members

bool `enableTimeout`

Enable or Disable timeout event trigger wakeup.

uint16_t `timeoutValue`

The timeout value that generates a wakeup event, the counter timer is incremented based on 64 times the CAN Bit Time unit.

bool `enableMatch`

Enable or Disable match event trigger wakeup.

flexcan_pn_match_source_t `matchSrc`

Selects the match source (ID and/or data match) to trigger wakeup.

uint8_t `matchNum`

The number of times a given message must match the predefined ID and/or data before generating a wakeup event, range in 0x1 ~ 0xFF.

flexcan_pn_match_mode_t `idMatchMode`

The ID match type.

flexcan_pn_match_mode_t `dataMatchMode`

The data match type.

uint32_t `idLower`

The ID target values 1 which used either for ID match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in ID match “range detection”.

uint32_t `idUpper`

The ID target values 2 which used only as the upper limit value in ID match “range detection” or used to store the ID mask in “equal to”.

uint8_t `lengthLower`

The lower limit for length of data bytes which used only in data match “range detection”. Range in 0x0 ~ 0x8.

uint8_t `lengthUpper`

The upper limit for length of data bytes which used only in data match “range detection”. Range in 0x0 ~ 0x8.

struct `_flexcan_rx_fifo_config`

#include <fsl_flexcan.h> FlexCAN Legacy Rx FIFO configuration structure.

Public Members

uint32_t `*idFilterTable`

Pointer to the FlexCAN Legacy Rx FIFO identifier filter table.

uint8_t idFilterNum

The FlexCAN Legacy Rx FIFO Filter elements quantity.

flexcan_rx_fifo_filter_type_t idFilterType

The FlexCAN Legacy Rx FIFO Filter type.

flexcan_rx_fifo_priority_t priority

The FlexCAN Legacy Rx FIFO receive priority.

struct *_flexcan_enhanced_rx_fifo_std_id_filter*

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

Public Members

uint32_t filterType

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1

CAN FD frame data length code (DLC), range see *_flexcan_fd_frame_length*, When the length ≤ 8 , it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use *DLC_LENGTH_DECODE(length)* macro to get the number of valid data bytes.

uint32_t std1

CAN Frame Type(DATA or REMOTE).

uint32_t rtr2

CAN Frame Identifier(STD or EXT format).

uint32_t std2

Substitute Remote request.

struct *_flexcan_enhanced_rx_fifo_ext_id_filter*

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

Public Members

uint32_t filterType

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1

CAN FD frame data length code (DLC), range see *_flexcan_fd_frame_length*, When the length ≤ 8 , it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use *DLC_LENGTH_DECODE(length)* macro to get the number of valid data bytes.

uint32_t std1

CAN Frame Type(DATA or REMOTE).

uint32_t rtr2

CAN Frame Identifier(STD or EXT format).

uint32_t std2

Substitute Remote request.

struct *_flexcan_enhanced_rx_fifo_config*

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO configuration structure.

Public Members

`uint32_t *idFilterTable`

Pointer to the FlexCAN Enhanced Rx FIFO identifier filter table, each table member occupies 32 bit word, table size should be equal to `idFilterNum`. There are two types of Enhanced Rx FIFO filter elements that can be stored in table : extended-ID filter element (1 word, occupies 1 table members) and standard-ID filter element (2 words, occupies 2 table members), the extended-ID filter element needs to be placed in front of the table.

`uint8_t idFilterPairNum`

`idFilterPairNum` is the Enhanced Rx FIFO identifier filter element pair numbers, each pair of filter elements occupies 2 words and can consist of one extended ID filter element or two standard ID filter elements.

`uint8_t extendIdFilterNum`

The number of extended ID filter element items in the FlexCAN enhanced Rx FIFO identifier filter table, each extended-ID filter element occupies 2 words, `extendIdFilterNum` need less than or equal to `idFilterPairNum`.

`uint8_t fifoWatermark`

$(\text{fifoWatermark} + 1)$ is the minimum number of CAN messages stored in the Enhanced RX FIFO which can trigger FIFO watermark interrupt or a DMA request.

`flexcan_efifo_dma_per_read_length_t dmaPerReadLength`

Define the length of each read of the Enhanced RX FIFO element by the DAM, see `_flexcan_fd_frame_length`.

`flexcan_rx_fifo_priority_t priority`

The FlexCAN Enhanced Rx FIFO receive priority.

`struct _flexcan_mb_transfer`

`#include <fsl_flexcan.h>` FlexCAN Message Buffer transfer.

Public Members

`flexcan_frame_t *frame`

The buffer of CAN Message to be transfer.

`uint8_t mbIdx`

The index of Message buffer used to transfer Message.

`struct _flexcan_fifo_transfer`

`#include <fsl_flexcan.h>` FlexCAN Rx FIFO transfer.

Public Members

`flexcan_fd_frame_t *framefd`

The buffer of CAN Message to be received from Enhanced Rx FIFO.

`flexcan_frame_t *frame`

The buffer of CAN Message to be received from Legacy Rx FIFO.

`size_t frameNum`

Number of CAN Message need to be received from Legacy or Enhanced Rx FIFO.

`struct _flexcan_handle`

`#include <fsl_flexcan.h>` FlexCAN handle structure.

Public Members

flexcan_transfer_callback_t callback

Callback function.

void *userData

FlexCAN callback function parameter.

flexcan_frame_t *volatile mbFrameBuf[CAN_WORD1_COUNT]

The buffer for received CAN data from Message Buffers.

flexcan_fd_frame_t *volatile mbFDFrameBuf[CAN_WORD1_COUNT]

The buffer for received CAN FD data from Message Buffers.

flexcan_frame_t *volatile rxFifoFrameBuf

The buffer for received CAN data from Legacy Rx FIFO.

flexcan_fd_frame_t *volatile rxFifoFDFrameBuf

The buffer for received CAN FD data from Enhanced Rx FIFO.

size_t rxFifoFrameNum

The number of CAN messages remaining to be received from Legacy or Enhanced Rx FIFO.

size_t rxFifoTransferTotalNum

Total CAN Message number need to be received from Legacy or Enhanced Rx FIFO.

volatile uint8_t mbState[CAN_WORD1_COUNT]

Message Buffer transfer state.

volatile uint8_t rxFifoState

Rx FIFO transfer state.

volatile uint32_t timestamp[CAN_WORD1_COUNT]

Mailbox transfer timestamp.

struct byteStatus

Public Members

bool byteIsRead

The byte n (0~3) was read or not. The type of error and which bit in byte (n) is affected by the error.

struct __unnamed14__

Public Members

uint32_t timestamp

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t length

CAN frame data length in bytes (Range: 0~8).

uint32_t type

CAN Frame Type(DATA or REMOTE).

uint32_t format

CAN Frame Identifier(STD or EXT format).

```
uint32_t __pad0__  
    Reserved.  
uint32_t idhit  
    CAN Rx FIFO filter hit id(This value is only used in Rx FIFO receive mode).  
struct __unnamed16__
```

Public Members

```
uint32_t id  
    CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD()  
    macro.  
uint32_t __pad0__  
    Reserved.  
union __unnamed18__
```

Public Members

```
struct __flexcan__frame  
  
struct __flexcan__frame  
  
struct __unnamed20__
```

Public Members

```
uint32_t dataWord0  
    CAN Frame payload word0.  
uint32_t dataWord1  
    CAN Frame payload word1.  
struct __unnamed22__
```

Public Members

```
uint8_t dataByte3  
    CAN Frame payload byte3.  
uint8_t dataByte2  
    CAN Frame payload byte2.  
uint8_t dataByte1  
    CAN Frame payload byte1.  
uint8_t dataByte0  
    CAN Frame payload byte0.  
uint8_t dataByte7  
    CAN Frame payload byte7.  
uint8_t dataByte6  
    CAN Frame payload byte6.
```

```
uint8_t dataByte5
    CAN Frame payload byte5.
uint8_t dataByte4
    CAN Frame payload byte4.
struct __unnamed24__
```

Public Members

```
uint32_t timestamp
    FlexCAN internal Free-Running Counter Time Stamp.
uint32_t length
    CAN FD frame data length code (DLC), range see _flexcan_fd_frame_length, When the
    length <= 8, it equal to the data length, otherwise the number of valid frame data is
    not equal to the length value. user can use DLC_LENGTH_DECODE(length) macro to
    get the number of valid data bytes.
uint32_t type
    CAN Frame Type(DATA only).
uint32_t format
    CAN Frame Identifier(STD or EXT format).
uint32_t srr
    Substitute Remote request.
uint32_t esi
    Error State Indicator.
uint32_t brs
    Bit Rate Switch.
uint32_t edl
    Extended Data Length.
struct __unnamed26__
```

Public Members

```
uint32_t id
    CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD()
    macro.
uint32_t __pad0__
    Reserved.
union __unnamed28__
```

Public Members

```
struct _flexcan_fd_frame
struct _flexcan_fd_frame
struct __unnamed30__
```

Public Members

uint32_t dataWord[16]

CAN FD Frame payload, 16 double word maximum.

struct ____unnamed32____

Public Members

uint8_t dataByte3

CAN Frame payload byte3.

uint8_t dataByte2

CAN Frame payload byte2.

uint8_t dataByte1

CAN Frame payload byte1.

uint8_t dataByte0

CAN Frame payload byte0.

uint8_t dataByte7

CAN Frame payload byte7.

uint8_t dataByte6

CAN Frame payload byte6.

uint8_t dataByte5

CAN Frame payload byte5.

uint8_t dataByte4

CAN Frame payload byte4.

union ____unnamed34____

Public Members

struct __flexcan__config

struct __flexcan__config

struct ____unnamed36____

Public Members

uint32_t baudRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t baudRateFD

FlexCAN FD bit rate in bps, for CANFD data phase.

struct ____unnamed38____

Public Members

uint32_t bitRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t bitRateFD

FlexCAN FD bit rate in bps, for CANFD data phase.

union ____unnamed40____

Public Members

struct __flexcan_pn_config

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range detection”.

struct __flexcan_pn_config

struct ____unnamed44____

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range detection”.

Public Members

uint32_t lowerWord0

CAN Frame payload word0.

uint32_t lowerWord1

CAN Frame payload word1.

struct ____unnamed46____

Public Members

uint8_t lowerByte3

CAN Frame payload byte3.

uint8_t lowerByte2

CAN Frame payload byte2.

uint8_t lowerByte1

CAN Frame payload byte1.

uint8_t lowerByte0

CAN Frame payload byte0.

uint8_t lowerByte7

CAN Frame payload byte7.

uint8_t lowerByte6

CAN Frame payload byte6.

uint8_t lowerByte5

CAN Frame payload byte5.

```
uint8_t lowerByte4
    CAN Frame payload byte4.
union __unnamed42__
```

Public Members

```
struct __flexcan_pn_config
    < The data target values 2 which used only as the upper limit value in data match
    "range
    detection" or used to store the data mask in "equal to".
struct __flexcan_pn_config
struct __unnamed48__
    < The data target values 2 which used only as the upper limit value in data match "range
    detection" or used to store the data mask in "equal to".
```

Public Members

```
uint32_t upperWord0
    CAN Frame payload word0.
uint32_t upperWord1
    CAN Frame payload word1.
struct __unnamed50__
```

Public Members

```
uint8_t upperByte3
    CAN Frame payload byte3.
uint8_t upperByte2
    CAN Frame payload byte2.
uint8_t upperByte1
    CAN Frame payload byte1.
uint8_t upperByte0
    CAN Frame payload byte0.
uint8_t upperByte7
    CAN Frame payload byte7.
uint8_t upperByte6
    CAN Frame payload byte6.
uint8_t upperByte5
    CAN Frame payload byte5.
uint8_t upperByte4
    CAN Frame payload byte4.
```

2.19 FlexCAN eDMA Driver

```
void FLEXCAN_TransferCreateHandleEDMA(CAN_Type *base, flexcan_edma_handle_t *handle,  
                                       flexcan_edma_transfer_callback_t callback, void  
                                       *userData, edma_handle_t *rxFifoEdmaHandle)
```

Initializes the FlexCAN handle, which is used in transactional functions.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxFifoEdmaHandle – User-requested DMA handle for Rx FIFO DMA transfer.

```
void FLEXCAN_PrepareTransfConfiguration(CAN_Type *base, flexcan_fifo_transfer_t *pFifoXfer,  
                                         edma_transfer_config_t *pEdmaConfig)
```

Prepares the eDMA transfer configuration for FLEXCAN Legacy RX FIFO.

This function prepares the eDMA transfer configuration structure according to FLEXCAN Legacy RX FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

```
status_t FLEXCAN_StartTransferDatafromRxFIFO(CAN_Type *base, flexcan_edma_handle_t  
                                              *handle, edma_transfer_config_t  
                                              *pEdmaConfig)
```

Start Transfer Data from the FLEXCAN Legacy Rx FIFO using eDMA.

This function to Update edma transfer configuration and Start eDMA transfer

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

```
status_t FLEXCAN_TransferReceiveFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle,  
                                           flexcan_fifo_transfer_t *pFifoXfer)
```

Receives the CAN Message from the Legacy Rx FIFO using eDMA.

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.

- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flex-can_fifo_transfer_t.

Return values

- kStatus__Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

status_t FLEXCAN__TransferGetReceiveFifoCountEMDA(CAN_Type *base, flexcan_edma_handle_t *handle, size_t *count)

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus__InvalidArgument – count is Invalid.
- kStatus__Success – Successfully return the count.

void FLEXCAN__TransferAbortReceiveFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle)

Aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

This function aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.

status_t FLEXCAN__TransferReceiveEnhancedFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle, flexcan_fifo_transfer_t *pFifoXfer)

Receives the CAN FD Message from the Enhanced Rx FIFO using eDMA.

This function receives the CAN FD Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flex-can_fifo_transfer_t.

Return values

- kStatus__Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

static inline *status_t* FLEXCAN__TransferGetReceiveEnhancedFifoCountEMDA(CAN_Type *base, flexcan_edma_handle_t *handle, size_t *count)

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `count` – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

FSL_FLEXCAN_EDMA_DRIVER_VERSION

FlexCAN EDMA driver version.

`typedef struct flexcan_edma_handle flexcan_edma_handle_t`

`typedef void (*flexcan_edma_transfer_callback_t)(CAN_Type *base, flexcan_edma_handle_t *handle, status_t status, void *userData)`

FlexCAN transfer callback function.

`struct __flexcan_edma_handle`

`#include <fsl_flexcan_edma.h>` FlexCAN eDMA handle.

Public Members

`flexcan_edma_transfer_callback_t` callback

Callback function.

`void *userData`

FlexCAN callback function parameter.

`edma_handle_t *rxFifoEdmaHandle`

The EDMA handler for Rx FIFO.

`volatile uint8_t rxFifoState`

Rx FIFO transfer state.

`size_t frameNum`

The number of messages that need to be received.

`flexcan_fd_frame_t *framefd`

Point to the buffer of CAN Message to be received from Enhanced Rx FIFO.

2.20 FlexIO: FlexIO Driver

2.21 FlexIO Driver

`void FLEXIO_GetDefaultConfig(flexio_config_t *userConfig)`

Gets the default configuration to configure the FlexIO module. The configuration can used directly to call the FLEXIO_Configure().

Example:

```
flexio_config_t config;
FLEXIO_GetDefaultConfig(&config);
```

Parameters

- userConfig – pointer to flexio_config_t structure

void FLEXIO_Init(FLEXIO_Type *base, const flexio_config_t *userConfig)

Configures the FlexIO with a FlexIO configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_GetDefaultConfig().

Example

```
flexio_config_t config = {  
    .enableFlexio = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false  
};  
FLEXIO_Configure(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- userConfig – pointer to flexio_config_t structure

void FLEXIO_Deinit(FLEXIO_Type *base)

Gates the FlexIO clock. Call this API to stop the FlexIO clock.

Note: After calling this API, call the FLEXIO_Init to use the FlexIO module.

Parameters

- base – FlexIO peripheral base address

uint32_t FLEXIO_GetInstance(FLEXIO_Type *base)

Get instance number for FLEXIO module.

Parameters

- base – FLEXIO peripheral base address.

void FLEXIO_Reset(FLEXIO_Type *base)

Resets the FlexIO module.

Parameters

- base – FlexIO peripheral base address

static inline void FLEXIO_Enable(FLEXIO_Type *base, bool enable)

Enables the FlexIO module operation.

Parameters

- base – FlexIO peripheral base address
- enable – true to enable, false to disable.

static inline uint32_t FLEXIO_ReadPinInput(FLEXIO_Type *base)

Reads the input data on each of the FlexIO pins.

Parameters

- base – FlexIO peripheral base address

Returns

FlexIO pin input data

```
static inline uint8_t FLEXIO_GetShifterState(FLEXIO_Type *base)
```

Gets the current state pointer for state mode use.

Parameters

- base – FlexIO peripheral base address

Returns

current State pointer

```
void FLEXIO_SetShifterConfig(FLEXIO_Type *base, uint8_t index, const flexio_shifter_config_t *shifterConfig)
```

Configures the shifter with the shifter configuration. The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {  
.timerSelect = 0,  
.timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,  
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,  
.pinPolarity = kFLEXIO_PinActiveLow,  
.shifterMode = kFLEXIO_ShifterModeTransmit,  
.inputSource = kFLEXIO_ShifterInputFromPin,  
.shifterStop = kFLEXIO_ShifterStopBitHigh,  
.shifterStart = kFLEXIO_ShifterStartBitLow  
};  
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Shifter index
- shifterConfig – Pointer to flexio_shifter_config_t structure

```
void FLEXIO_SetTimerConfig(FLEXIO_Type *base, uint8_t index, const flexio_timer_config_t *timerConfig)
```

Configures the timer with the timer configuration. The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

Example

```
flexio_timer_config_t config = {  
.triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFToNSTAT(0),  
.triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,  
.triggerSource = kFLEXIO_TimerTriggerSourceInternal,  
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,  
.pinSelect = 0,  
.pinPolarity = kFLEXIO_PinActiveHigh,  
.timerMode = kFLEXIO_TimerModeDual8BitBaudBit,  
.timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,  
.timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput,  
.timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,  
.timerDisable = kFLEXIO_TimerDisableOnTimerCompare,  
.timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,  
.timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,  
.timerStart = kFLEXIO_TimerStartBitEnabled  
};  
FLEXIO_SetTimerConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Timer index
- timerConfig – Pointer to the flexio_timer_config_t structure

```
static inline void FLEXIO_SetClockMode(FLEXIO_Type *base, uint8_t index,  
                                       flexio_timer_decrement_source_t clocksource)
```

This function set the value of the prescaler on flexio channels.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- index – Timer index
- clocksource – Set clock value

```
static inline void FLEXIO_EnableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

```
static inline void FLEXIO_DisableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Disables the shifter status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

```
static inline void FLEXIO_EnableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter error interrupt. The interrupt generates when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_DisableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)

Disables the shifter error interrupt. The interrupt won't generate when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_EnableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Enables the timer status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_DisableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Disables the timer status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline uint32_t FLEXIO_GetShifterStatusFlags(FLEXIO_Type *base)

Gets the shifter status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter status flags

static inline void FLEXIO_ClearShifterStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter status flags.

Note: For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address

- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetShifterErrorFlags(FLEXIO_Type *base)

Gets the shifter error flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter error flags

static inline void FLEXIO_ClearShifterErrorFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter error flags.

Note: For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetTimerStatusFlags(FLEXIO_Type *base)

Gets the timer status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Timer status flags

static inline void FLEXIO_ClearTimerStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the timer status flags.

Note: For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_EnableShifterStatusDMA(FLEXIO_Type *base, uint32_t mask, bool enable)

Enables/disables the shifter status DMA. The DMA request generates when the corresponding SSF is set.

Note: For multiple shifter status DMA enables, for example, calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

- enable – True to enable, false to disable.

uint32_t FLEXIO_GetShifterBufferAddress(FLEXIO_Type *base, flexio_shifter_buffer_type_t type, uint8_t index)

Gets the shifter buffer address for the DMA transfer usage.

Parameters

- base – FlexIO peripheral base address
- type – Shifter type of flexio_shifter_buffer_type_t
- index – Shifter index

Returns

Corresponding shifter buffer index

status_t FLEXIO_RegisterHandleIRQ(void *base, void *handle, flexio_isr_t isr)

Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- handle – Pointer to the handler for FlexIO simulated peripheral.
- isr – FlexIO simulated peripheral interrupt handler.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_UnregisterHandleIRQ(void *base)

Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

static inline void FLEXIO_ClearPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 0.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_SetPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 1.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_TogglePortOutput(FLEXIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple FLEXIO pins.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_PinWrite(FLEXIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the FLEXIO pins to the logic 1 or 0.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- output – FLEXIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void FLEXIO_EnablePinOutput(FLEXIO_Type *base, uint32_t pin)

Enables the FLEXIO output pin function.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

static inline uint32_t FLEXIO_PinRead(FLEXIO_Type *base, uint32_t pin)

Reads the current input value of the FLEXIO pin.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline uint32_t FLEXIO_GetPinStatus(FLEXIO_Type *base, uint32_t pin)

Gets the FLEXIO input pin status.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input status

- 0: corresponding pin input capture no status.
- 1: corresponding pin input capture rising or falling edge.

static inline void FLEXIO_SetPinLevel(FLEXIO_Type *base, uint8_t pin, bool level)

Sets the FLEXIO output pin level.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.
- level – FlexIO output pin level to set, can be either 0 or 1.

static inline bool FLEXIO_GetPinOverride(const FLEXIO_Type *const base, uint8_t pin)

Gets the enabled status of a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address

- pin – FlexIO pin number.

Return values

FlexIO – port enabled status

- 0: corresponding output pin is in disabled state.
- 1: corresponding output pin is in enabled state.

static inline void FLEXIO_ConfigPinOverride(FLEXIO_Type *base, uint8_t pin, bool enabled)
Enables or disables a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – Flexio pin number.
- enabled – Enable or disable the FlexIO pin.

static inline void FLEXIO_ClearPortStatus(FLEXIO_Type *base, uint32_t mask)
Clears the multiple FLEXIO input pins status.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

FSL_FLEXIO_DRIVER_VERSION
FlexIO driver version.

enum _flexio_timer_trigger_polarity
Define time of timer trigger polarity.

Values:

enumerator kFLEXIO_TimerTriggerPolarityActiveHigh
Active high.

enumerator kFLEXIO_TimerTriggerPolarityActiveLow
Active low.

enum _flexio_timer_trigger_source
Define type of timer trigger source.

Values:

enumerator kFLEXIO_TimerTriggerSourceExternal
External trigger selected.

enumerator kFLEXIO_TimerTriggerSourceInternal
Internal trigger selected.

enum _flexio_pin_config
Define type of timer/shifter pin configuration.

Values:

enumerator kFLEXIO_PinConfigOutputDisabled
Pin output disabled.

enumerator kFLEXIO_PinConfigOpenDrainOrBidirection
Pin open drain or bidirectional output enable.

enumerator kFLEXIO_PinConfigBidirectionOutputData
Pin bidirectional output data.

enumerator kFLEXIO_PinConfigOutput

Pin output.

enum _flexio_pin_polarity

Definition of pin polarity.

Values:

enumerator kFLEXIO_PinActiveHigh

Active high.

enumerator kFLEXIO_PinActiveLow

Active low.

enum _flexio_timer_mode

Define type of timer work mode.

Values:

enumerator kFLEXIO_TimerModeDisabled

Timer Disabled.

enumerator kFLEXIO_TimerModeDual8BitBaudBit

Dual 8-bit counters baud/bit mode.

enumerator kFLEXIO_TimerModeDual8BitPWM

Dual 8-bit counters PWM mode.

enumerator kFLEXIO_TimerModeSingle16Bit

Single 16-bit counter mode.

enumerator kFLEXIO_TimerModeDual8BitPWMLow

Dual 8-bit counters PWM Low mode.

enum _flexio_timer_output

Define type of timer initial output or timer reset condition.

Values:

enumerator kFLEXIO_TimerOutputOneNotAffectedByReset

Logic one when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputZeroNotAffectedByReset

Logic zero when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputOneAffectedByReset

Logic one when enabled and on timer reset.

enumerator kFLEXIO_TimerOutputZeroAffectedByReset

Logic zero when enabled and on timer reset.

enum _flexio_timer_decrement_source

Define type of timer decrement.

Values:

enumerator kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput

Decrement counter on FlexIO clock, Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput

Decrement counter on Trigger input (both edges), Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnPinInputShiftPinInput

Decrement counter on Pin input (both edges), Shift clock equals Pin input.

enumerator kFLEXIO__TimerDecSrcOnTriggerInputShiftTriggerInput
Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

enum _flexio_timer_reset_condition

Define type of timer reset condition.

Values:

enumerator kFLEXIO__TimerResetNever
Timer never reset.

enumerator kFLEXIO__TimerResetOnTimerPinEqualToTimerOutput
Timer reset on Timer Pin equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerTriggerEqualToTimerOutput
Timer reset on Timer Trigger equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerPinRisingEdge
Timer reset on Timer Pin rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerRisingEdge
Timer reset on Trigger rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerBothEdge
Timer reset on Trigger rising or falling edge.

enum _flexio_timer_disable_condition

Define type of timer disable condition.

Values:

enumerator kFLEXIO__TimerDisableNever
Timer never disabled.

enumerator kFLEXIO__TimerDisableOnPreTimerDisable
Timer disabled on Timer N-1 disable.

enumerator kFLEXIO__TimerDisableOnTimerCompare
Timer disabled on Timer compare.

enumerator kFLEXIO__TimerDisableOnTimerCompareTriggerLow
Timer disabled on Timer compare and Trigger Low.

enumerator kFLEXIO__TimerDisableOnPinBothEdge
Timer disabled on Pin rising or falling edge.

enumerator kFLEXIO__TimerDisableOnPinBothEdgeTriggerHigh
Timer disabled on Pin rising or falling edge provided Trigger is high.

enumerator kFLEXIO__TimerDisableOnTriggerFallingEdge
Timer disabled on Trigger falling edge.

enum _flexio_timer_enable_condition

Define type of timer enable condition.

Values:

enumerator kFLEXIO__TimerEnabledAlways
Timer always enabled.

enumerator kFLEXIO__TimerEnableOnPrevTimerEnable
Timer enabled on Timer N-1 enable.

enumerator kFLEXIO__TimerEnableOnTriggerHigh
Timer enabled on Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerHighPinHigh
Timer enabled on Trigger high and Pin high.

enumerator kFLEXIO__TimerEnableOnPinRisingEdge
Timer enabled on Pin rising edge.

enumerator kFLEXIO__TimerEnableOnPinRisingEdgeTriggerHigh
Timer enabled on Pin rising edge and Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerRisingEdge
Timer enabled on Trigger rising edge.

enumerator kFLEXIO__TimerEnableOnTriggerBothEdge
Timer enabled on Trigger rising or falling edge.

enum _flexio_timer_stop_bit_condition
Define type of timer stop bit generate condition.

Values:

enumerator kFLEXIO__TimerStopBitDisabled
Stop bit disabled.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompare
Stop bit is enabled on timer compare.

enumerator kFLEXIO__TimerStopBitEnableOnTimerDisable
Stop bit is enabled on timer disable.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompareDisable
Stop bit is enabled on timer compare and timer disable.

enum _flexio_timer_start_bit_condition
Define type of timer start bit generate condition.

Values:

enumerator kFLEXIO__TimerStartBitDisabled
Start bit disabled.

enumerator kFLEXIO__TimerStartBitEnabled
Start bit enabled.

enum _flexio_timer_output_state
FlexIO as PWM channel output state.

Values:

enumerator kFLEXIO__PwmLow
The output state of PWM channel is low

enumerator kFLEXIO__PwmHigh
The output state of PWM channel is high

enum _flexio_shifter_timer_polarity
Define type of timer polarity for shifter control.

Values:

enumerator kFLEXIO__ShifterTimerPolarityOnPositive
Shift on positive edge of shift clock.

enumerator kFLEXIO__ShifterTimerPolarityOnNegative
Shift on negative edge of shift clock.

enum _flexio__shifter__mode

Define type of shifter working mode.

Values:

enumerator kFLEXIO__ShifterDisabled
Shifter is disabled.

enumerator kFLEXIO__ShifterModeReceive
Receive mode.

enumerator kFLEXIO__ShifterModeTransmit
Transmit mode.

enumerator kFLEXIO__ShifterModeMatchStore
Match store mode.

enumerator kFLEXIO__ShifterModeMatchContinuous
Match continuous mode.

enumerator kFLEXIO__ShifterModeState
SHIFTBUF contents are used for storing programmable state attributes.

enumerator kFLEXIO__ShifterModeLogic
SHIFTBUF contents are used for implementing programmable logic look up table.

enum _flexio__shifter__input__source

Define type of shifter input source.

Values:

enumerator kFLEXIO__ShifterInputFromPin
Shifter input from pin.

enumerator kFLEXIO__ShifterInputFromNextShifterOutput
Shifter input from Shifter N+1.

enum _flexio__shifter__stop__bit

Define of STOP bit configuration.

Values:

enumerator kFLEXIO__ShifterStopBitDisable
Disable shifter stop bit.

enumerator kFLEXIO__ShifterStopBitLow
Set shifter stop bit to logic low level.

enumerator kFLEXIO__ShifterStopBitHigh
Set shifter stop bit to logic high level.

enum _flexio__shifter__start__bit

Define type of START bit configuration.

Values:

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnEnable
Disable shifter start bit, transmitter loads data on enable.

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnShift
Disable shifter start bit, transmitter loads data on first shift.

enumerator kFLEXIO__ShifterStartBitLow
Set shifter start bit to logic low level.

enumerator kFLEXIO__ShifterStartBitHigh
Set shifter start bit to logic high level.

enum __flexio_shifter_buffer_type
Define FlexIO shifter buffer type.

Values:

enumerator kFLEXIO__ShifterBuffer
Shifter Buffer N Register.

enumerator kFLEXIO__ShifterBufferBitSwapped
Shifter Buffer N Bit Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferByteSwapped
Shifter Buffer N Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferBitByteSwapped
Shifter Buffer N Bit Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleByteSwapped
Shifter Buffer N Nibble Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferHalfWordSwapped
Shifter Buffer N Half Word Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleSwapped
Shifter Buffer N Nibble Swapped Register.

enum __flexio_gpio_direction
FLEXIO gpio direction definition.

Values:

enumerator kFLEXIO__DigitalInput
Set current pin as digital input

enumerator kFLEXIO__DigitalOutput
Set current pin as digital output

enum __flexio_pin_input_config
FLEXIO gpio input config.

Values:

enumerator kFLEXIO__InputInterruptDisabled
Interrupt request is disabled.

enumerator kFLEXIO__InputInterruptEnable
Interrupt request is enable.

enumerator kFLEXIO__FlagRisingEdgeEnable
Input pin flag on rising edge.

enumerator kFLEXIO__FlagFallingEdgeEnable
Input pin flag on falling edge.

typedef enum __flexio_timer_trigger_polarity flexio_timer_trigger_polarity_t
Define time of timer trigger polarity.

`typedef enum _flexio_timer_trigger_source flexio_timer_trigger_source_t`
Define type of timer trigger source.

`typedef enum _flexio_pin_config flexio_pin_config_t`
Define type of timer/shifter pin configuration.

`typedef enum _flexio_pin_polarity flexio_pin_polarity_t`
Definition of pin polarity.

`typedef enum _flexio_timer_mode flexio_timer_mode_t`
Define type of timer work mode.

`typedef enum _flexio_timer_output flexio_timer_output_t`
Define type of timer initial output or timer reset condition.

`typedef enum _flexio_timer_decrement_source flexio_timer_decrement_source_t`
Define type of timer decrement.

`typedef enum _flexio_timer_reset_condition flexio_timer_reset_condition_t`
Define type of timer reset condition.

`typedef enum _flexio_timer_disable_condition flexio_timer_disable_condition_t`
Define type of timer disable condition.

`typedef enum _flexio_timer_enable_condition flexio_timer_enable_condition_t`
Define type of timer enable condition.

`typedef enum _flexio_timer_stop_bit_condition flexio_timer_stop_bit_condition_t`
Define type of timer stop bit generate condition.

`typedef enum _flexio_timer_start_bit_condition flexio_timer_start_bit_condition_t`
Define type of timer start bit generate condition.

`typedef enum _flexio_timer_output_state flexio_timer_output_state_t`
FlexIO as PWM channel output state.

`typedef enum _flexio_shifter_timer_polarity flexio_shifter_timer_polarity_t`
Define type of timer polarity for shifter control.

`typedef enum _flexio_shifter_mode flexio_shifter_mode_t`
Define type of shifter working mode.

`typedef enum _flexio_shifter_input_source flexio_shifter_input_source_t`
Define type of shifter input source.

`typedef enum _flexio_shifter_stop_bit flexio_shifter_stop_bit_t`
Define of STOP bit configuration.

`typedef enum _flexio_shifter_start_bit flexio_shifter_start_bit_t`
Define type of START bit configuration.

`typedef enum _flexio_shifter_buffer_type flexio_shifter_buffer_type_t`
Define FlexIO shifter buffer type.

`typedef struct _flexio_config flexio_config_t`
Define FlexIO user configuration structure.

`typedef struct _flexio_timer_config flexio_timer_config_t`
Define FlexIO timer configuration structure.

`typedef struct _flexio_shifter_config flexio_shifter_config_t`
Define FlexIO shifter configuration structure.

typedef enum *flexio_gpio_direction* flexio_gpio_direction_t
FLEXIO gpio direction definition.

typedef enum *flexio_pin_input_config* flexio_pin_input_config_t
FLEXIO gpio input config.

typedef struct *flexio_gpio_config* flexio_gpio_config_t
The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

typedef void (*flexio_isr_t)(void *base, void *handle)
typedef for FlexIO simulated driver interrupt handler.

FLEXIO_Type *const s_flexioBases[]
Pointers to flexio bases for each instance.

const clock_ip_name_t s_flexioClocks[]
Pointers to flexio clocks for each instance.

void FLEXIO_SetPinConfig(FLEXIO_Type *base, uint32_t pin, *flexio_gpio_config_t* *config)
Configure a FLEXIO pin used by the board.

To Config the FLEXIO PIN, define a pin configuration, as either input or output, in the user file. Then, call the FLEXIO_SetPinConfig() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalInput,
    0U,
    kFLEXIO_FlagRisingEdgeEnable | kFLEXIO_InputInterruptEnable,
}
Define a digital output pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalOutput,
    0U,
    0U
}
```

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- config – FLEXIO pin configuration pointer.

FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x)
Calculate FlexIO timer trigger.

FLEXIO_TIMER_TRIGGER_SEL_SHIFThSTAT(x)

FLEXIO_TIMER_TRIGGER_SEL_TIMn(x)

struct flexio_config_
#include <fsl_flexio.h> Define FlexIO user configuration structure.

Public Members

bool enableFlexio

Enable/disable FlexIO module

bool enableInDoze

Enable/disable FlexIO operation in doze mode

bool enableInDebug

Enable/disable FlexIO operation in debug mode

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

struct *_flexio_timer_config*

#include <fsl_flexio.h> Define FlexIO timer configuration structure.

Public Members

uint32_t triggerSelect

The internal trigger selection number using MACROs.

flexio_timer_trigger_polarity_t triggerPolarity

Trigger Polarity.

flexio_timer_trigger_source_t triggerSource

Trigger Source, internal (see 'trgsel') or external.

flexio_pin_config_t pinConfig

Timer Pin Configuration.

uint32_t pinSelect

Timer Pin number Select.

flexio_pin_polarity_t pinPolarity

Timer Pin Polarity.

flexio_timer_mode_t timerMode

Timer work Mode.

flexio_timer_output_t timerOutput

Configures the initial state of the Timer Output and whether it is affected by the Timer reset.

flexio_timer_decrement_source_t timerDecrement

Configures the source of the Timer decrement and the source of the Shift clock.

flexio_timer_reset_condition_t timerReset

Configures the condition that causes the timer counter (and optionally the timer output) to be reset.

flexio_timer_disable_condition_t timerDisable

Configures the condition that causes the Timer to be disabled and stop decrementing.

flexio_timer_enable_condition_t timerEnable

Configures the condition that causes the Timer to be enabled and start decrementing.

flexio_timer_stop_bit_condition_t timerStop

Timer STOP Bit generation.

flexio_timer_start_bit_condition_t timerStart

Timer STRAT Bit generation.

uint32_t timerCompare

Value for Timer Compare N Register.

struct *_flexio_shifter_config*

#include <fsl_flexio.h> Define FlexIO shifter configuration structure.

Public Members

uint32_t timerSelect

Selects which Timer is used for controlling the logic/shift register and generating the Shift clock.

flexio_shifter_timer_polarity_t timerPolarity

Timer Polarity.

flexio_pin_config_t pinConfig

Shifter Pin Configuration.

uint32_t pinSelect

Shifter Pin number Select.

flexio_pin_polarity_t pinPolarity

Shifter Pin Polarity.

flexio_shifter_mode_t shifterMode

Configures the mode of the Shifter.

uint32_t parallelWidth

Configures the parallel width when using parallel mode.

flexio_shifter_input_source_t inputSource

Selects the input source for the shifter.

flexio_shifter_stop_bit_t shifterStop

Shifter STOP bit.

flexio_shifter_start_bit_t shifterStart

Shifter START bit.

struct *_flexio_gpio_config*

#include <fsl_flexio.h> The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

Public Members

flexio_gpio_direction_t pinDirection

FLEXIO pin direction, input or output

uint8_t outputLogic

Set a default output logic, which has no use in input

uint8_t inputConfig

Set an input config

2.22 FlexIO eDMA SPI Driver

```
status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA(FLEXIO_SPI_Type *base,  
                                                    flexio_spi_master_edma_handle_t  
                                                    *handle,  
                                                    flexio_spi_master_edma_transfer_callback_t  
                                                    callback, void *userData,  
                                                    edma_handle_t *txHandle,  
                                                    edma_handle_t *rxHandle)
```

Initializes the FlexIO SPI master eDMA handle.

This function initializes the FlexIO SPI master eDMA handle which can be used for other FlexIO SPI master transactional APIs. For a specified FlexIO SPI instance, call this API once to get the initialized handle.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

```
status_t FLEXIO_SPI_MasterTransferEDMA(FLEXIO_SPI_Type *base,  
                                         flexio_spi_master_edma_handle_t *handle,  
                                         flexio_spi_transfer_t *xfer)
```

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call FLEXIO_SPI_MasterGetTransferCountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – FlexIO SPI is not idle, is running another transfer.

```
void FLEXIO_SPI_MasterTransferAbortEDMA(FLEXIO_SPI_Type *base,
                                         flexio_spi_master_edma_handle_t *handle)
```

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – FlexIO SPI eDMA handle pointer.

```
status_t FLEXIO_SPI_MasterTransferGetCountEDMA(FLEXIO_SPI_Type *base,
                                                flexio_spi_master_edma_handle_t *handle,
                                                size_t *count)
```

Gets the number of bytes transferred so far using FlexIO SPI master eDMA.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – FlexIO SPI eDMA handle pointer.
- count – Number of bytes transferred so far by the non-blocking transaction.

```
static inline void FLEXIO_SPI_SlaveTransferCreateHandleEDMA(FLEXIO_SPI_Type *base,
                                                           flexio_spi_slave_edma_handle_t
                                                           *handle,
                                                           flexio_spi_slave_edma_transfer_callback_t
                                                           callback, void *userData,
                                                           edma_handle_t *txHandle,
                                                           edma_handle_t *rxHandle)
```

Initializes the FlexIO SPI slave eDMA handle.

This function initializes the FlexIO SPI slave eDMA handle.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – Pointer to *flexio_spi_slave_edma_handle_t* structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.

```
status_t FLEXIO_SPI_SlaveTransferEDMA(FLEXIO_SPI_Type *base,
                                       flexio_spi_slave_edma_handle_t *handle,
                                       flexio_spi_transfer_t *xfer)
```

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call *FLEXIO_SPI_SlaveGetTransferCountEDMA* to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – Pointer to *flexio_spi_slave_edma_handle_t* structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_FLEXIO_SPI_Busy` – FlexIO SPI is not idle, is running another transfer.

```
static inline void FLEXIO_SPI_SlaveTransferAbortEDMA(FLEXIO_SPI_Type *base,  
                                                    flexio_spi_slave_edma_handle_t  
                                                    *handle)
```

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- `base` – Pointer to `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to `flexio_spi_slave_edma_handle_t` structure to store the transfer state.

```
static inline status_t FLEXIO_SPI_SlaveTransferGetCountEDMA(FLEXIO_SPI_Type *base,  
                                                         flexio_spi_slave_edma_handle_t  
                                                         *handle, size_t *count)
```

Gets the number of bytes transferred so far using FlexIO SPI slave eDMA.

Parameters

- `base` – Pointer to `FLEXIO_SPI_Type` structure.
- `handle` – FlexIO SPI eDMA handle pointer.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

`FSL_FLEXIO_SPI_EDMA_DRIVER_VERSION`

FlexIO SPI EDMA driver version.

```
typedef struct flexio_spi_master_edma_handle flexio_spi_master_edma_handle_t  
typedef for flexio_spi_master_edma_handle_t in advance.
```

```
typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t  
Slave handle is the same with master handle.
```

```
typedef void (*flexio_spi_master_edma_transfer_callback_t)(FLEXIO_SPI_Type *base,  
flexio_spi_master_edma_handle_t *handle, status_t status, void *userData)
```

FlexIO SPI master callback for finished transmit.

```
typedef void (*flexio_spi_slave_edma_transfer_callback_t)(FLEXIO_SPI_Type *base,  
flexio_spi_slave_edma_handle_t *handle, status_t status, void *userData)
```

FlexIO SPI slave callback for finished transmit.

```
struct flexio_spi_master_edma_handle
```

```
#include <fsl_flexio_spi_edma.h> FlexIO SPI eDMA transfer handle, users should not touch  
the content of the handle.
```

Public Members

`size_t` `transferSize`

Total bytes to be transferred.

`uint8_t` `nbytes`

eDMA minor byte transfer count initially configured.

bool txInProgress
 Send transfer in progress
 bool rxInProgress
 Receive transfer in progress
 edma_handle_t *txHandle
 DMA handler for SPI send
 edma_handle_t *rxHandle
 DMA handler for SPI receive
 flexio_spi_master_edma_transfer_callback_t callback
 Callback for SPI DMA transfer
 void *userData
 User Data for SPI DMA callback

2.23 FlexIO eDMA UART Driver

```
status_t FLEXIO_UART_TransferCreateHandleEDMA(FLEXIO_UART_Type *base,
                                              flexio_uart_edma_handle_t *handle,
                                              flexio_uart_edma_transfer_callback_t
                                              callback, void *userData, edma_handle_t
                                              *txEdmaHandle, edma_handle_t
                                              *rxEdmaHandle)
```

Initializes the UART handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_UART_Type.
- handle – Pointer to flexio_uart_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxEdmaHandle – User requested DMA handle for RX DMA transfer.
- txEdmaHandle – User requested DMA handle for TX DMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

```
status_t FLEXIO_UART_TransferSendEDMA(FLEXIO_UART_Type *base,
                                       flexio_uart_edma_handle_t *handle,
                                       flexio_uart_transfer_t *xfer)
```

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent out, the send callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – UART handle pointer.
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXIO_UART_TxBusy – Previous transfer on going.

status_t FLEXIO_UART_TransferReceiveEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle,
 flexio_uart_transfer_t *xfer)

Receives data using eDMA.

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_UART_RxBusy – Previous transfer on going.

void FLEXIO_UART_TransferAbortSendEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle)

Aborts the sent data which using eDMA.

This function aborts sent data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

void FLEXIO_UART_TransferAbortReceiveEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle)

Aborts the receive data which using eDMA.

This function aborts the receive data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

status_t FLEXIO_UART_TransferGetSendCountEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle,
 size_t *count)

Gets the number of bytes sent out.

This function gets the number of bytes sent out.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- count – Number of bytes sent so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus_Success – Successfully return the count.


```
status_t FLEXIO_UART_TransferGetReceiveCountEDMA(FLEXIO_UART_Type *base,
                                                flexio_uart_edma_handle_t *handle,
                                                size_t *count)
```

Gets the number of bytes received.

This function gets the number of bytes received.

Parameters

- base – Pointer to *FLEXIO_UART_Type*
- handle – Pointer to *flexio_uart_edma_handle_t* structure
- count – Number of bytes received so far by the non-blocking transaction.

Return values

- *kStatus_NoTransferInProgress* – transfer has finished or no transfer in progress.
- *kStatus_Success* – Successfully return the count.

```
FSL_FLEXIO_UART_EDMA_DRIVER_VERSION
```

FlexIO UART EDMA driver version.

```
typedef struct flexio_uart_edma_handle flexio_uart_edma_handle_t
```

```
typedef void (*flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base,
flexio_uart_edma_handle_t *handle, status_t status, void *userData)
```

UART transfer callback function.

```
struct flexio_uart_edma_handle
```

```
#include <fsl_flexio_uart_edma.h> UART eDMA handle.
```

Public Members

```
flexio_uart_edma_transfer_callback_t callback
```

Callback function.

```
void *userData
```

UART callback function parameter.

```
size_t txDataSizeAll
```

Total bytes to be sent.

```
size_t rxDataSizeAll
```

Total bytes to be received.

```
edma_handle_t *txEdmaHandle
```

The eDMA TX channel used.

```
edma_handle_t *rxEdmaHandle
```

The eDMA RX channel used.

```
uint8_t nbytes
```

eDMA minor byte transfer count initially configured.

```
volatile uint8_t txState
```

TX transfer state.

```
volatile uint8_t rxState
```

RX transfer state

2.24 FlexIO I2C Master Driver

status_t FLEXIO_I2C_CheckForBusyBus(*FLEXIO_I2C_Type* *base)

Make sure the bus isn't already pulled down.

Check the FLEXIO pin status to see whether either of SDA and SCL pin is pulled down.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure..

Return values

- kStatus_Success –
- kStatus_FLEXIO_I2C_Busy –

status_t FLEXIO_I2C_MasterInit(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_config_t* *masterConfig, *uint32_t* srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.

Example

```
FLEXIO_I2C_Type base = {  
.flexioBase = FLEXIO,  
.SDAPinIndex = 0,  
.SCLPinIndex = 1,  
.shifterIndex = {0,1},  
.timerIndex = {0,1}  
};  
flexio_i2c_master_config_t config = {  
.enableInDoze = false,  
.enableInDebug = true,  
.enableFastAccess = false,  
.baudRate_Bps = 100000  
};  
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- masterConfig – Pointer to flexio_i2c_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Initialization successful
- kStatus_InvalidArgument – The source clock exceed upper range limitation

void FLEXIO_I2C_MasterDeinit(*FLEXIO_I2C_Type* *base)

De-initializes the FlexIO I2C master peripheral. Calling this API Resets the FlexIO I2C master shifer and timer config, module can't work unless the FLEXIO_I2C_MasterInit is called.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterGetDefaultConfig(*flexio_i2c_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO module. The configuration can be used directly for calling the FLEXIO_I2C_MasterInit().

Example:

```
flexio_i2c_master_config_t config;
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – Pointer to flexio_i2c_master_config_t structure.

static inline void FLEXIO_I2C_MasterEnable(*FLEXIO_I2C_Type* *base, bool enable)

Enables/disables the FlexIO module operation.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – Pass true to enable module, false does not have any effect.

uint32_t FLEXIO_I2C_MasterGetStatusFlags(*FLEXIO_I2C_Type* *base)

Gets the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure

Returns

Status flag, use status flag to AND _flexio_i2c_master_status_flags can get the related status.

void FLEXIO_I2C_MasterClearStatusFlags(*FLEXIO_I2C_Type* *base, uint32_t mask)

Clears the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
 - kFLEXIO_I2C_RxFullFlag
 - kFLEXIO_I2C_ReceiveNakFlag

void FLEXIO_I2C_MasterEnableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)

Enables the FlexIO i2c master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source. Currently only one interrupt request source:
 - kFLEXIO_I2C_TransferCompleteInterruptEnable

void FLEXIO_I2C_MasterDisableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)

Disables the FlexIO I2C master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source.

void FLEXIO_I2C_MasterSetBaudRate(*FLEXIO_I2C_Type* *base, uint32_t baudRate_Bps,
uint32_t srcClock_Hz)

Sets the FlexIO I2C master transfer baudrate.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- baudRate_Bps – the baud rate value in HZ

- srcClock_Hz – source clock in HZ

void FLEXIO_I2C_MasterStart(*FLEXIO_I2C_Type* *base, uint8_t address, *flexio_i2c_direction_t* direction)

Sends START + 7-bit address to the bus.

Note: This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO_I2C_RxFullFlag status is asserted before calling this API.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- address – 7-bit address.
- direction – transfer direction. This parameter is one of the values in flexio_i2c_direction_t:
 - kFLEXIO_I2C_Write: Transmit
 - kFLEXIO_I2C_Read: Receive

void FLEXIO_I2C_MasterStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterRepeatedStart(*FLEXIO_I2C_Type* *base)

Sends the repeated start signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterAbortStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal when transfer is still on-going.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterEnableAck(*FLEXIO_I2C_Type* *base, bool enable)

Configures the sent ACK/NAK for the following byte.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – True to configure send ACK, false configure to send NAK.

status_t FLEXIO_I2C_MasterSetTransferCount(*FLEXIO_I2C_Type* *base, uint16_t count)

Sets the number of bytes to be transferred from a start signal to a stop signal.

Note: Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

- `count` – Number of bytes need to be transferred from a start signal to a re-start/stop signal

Return values

- `kStatus_Success` – Successfully configured the count.
- `kStatus_InvalidArgument` – Input argument is invalid.

```
static inline void FLEXIO_I2C_MasterWriteByte(FLEXIO_I2C_Type *base, uint32_t data)
```

Writes one byte of data to the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the `TxEmptyFlag` is asserted before calling this API.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `data` – a byte of data.

```
static inline uint8_t FLEXIO_I2C_MasterReadByte(FLEXIO_I2C_Type *base)
```

Reads one byte of data from the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.

Returns

data byte read.

```
status_t FLEXIO_I2C_MasterWriteBlocking(FLEXIO_I2C_Type *base, const uint8_t *txBuff,  
                                         uint8_t txSize)
```

Sends a buffer of data in bytes.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `txBuff` – The data bytes to send.
- `txSize` – The number of data bytes to send.

Return values

- `kStatus_Success` – Successfully write data.
- `kStatus_FLEXIO_I2C_Nak` – Receive NAK during writing data.
- `kStatus_FLEXIO_I2C_Timeout` – Timeout polling status flags.

```
status_t FLEXIO_I2C_MasterReadBlocking(FLEXIO_I2C_Type *base, uint8_t *rxBuff, uint8_t  
                                         rxSize)
```

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- rxBuff – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus_Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

status_t FLEXIO_I2C_MasterTransferBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.
- xfer – pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

status_t FLEXIO_I2C_MasterTransferCreateHandle(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_callback_t
callback, void *userData)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure to store the transfer state.
- callback – Pointer to user callback function.
- userData – User param passed to the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/isr table out of range.

status_t FLEXIO_I2C_MasterTransferNonBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_t *xfer)

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: The API returns immediately after the transfer initiates. Call FLEXIO_I2C_MasterTransferGetCount to poll the transfer status to check whether the

transfer is finished. If the return status is not kStatus_FLEXIO_I2C_Busy, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state
- xfer – pointer to flexio_i2c_master_transfer_t structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_FLEXIO_I2C_Busy – FlexIO I2C is not idle, is running another transfer.

status_t FLEXIO_I2C_MasterTransferGetCount(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle, size_t
*count)

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.
- kStatus_Success – Successfully return the count.

void FLEXIO_I2C_MasterTransferAbort(*FLEXIO_I2C_Type* *base, flexio_i2c_master_handle_t
*handle)

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state

void FLEXIO_I2C_MasterTransferHandleIRQ(void *i2cType, void *i2cHandle)

Master interrupt handler.

Parameters

- i2cType – Pointer to FLEXIO_I2C_Type structure
- i2cHandle – Pointer to flexio_i2c_master_transfer_t structure

FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION

FlexIO I2C transfer status.

Values:

enumerator kStatus_FLEXIO_I2C_Busy

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Idle

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Nak

NAK received during transfer.

enumerator kStatus_FLEXIO_I2C_Timeout

Timeout polling status flags.

enum _flexio_i2c_master_interrupt

Define FlexIO I2C master interrupt mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyInterruptEnable

Tx buffer empty interrupt enable.

enumerator kFLEXIO_I2C_RxFullInterruptEnable

Rx buffer full interrupt enable.

enum _flexio_i2c_master_status_flags

Define FlexIO I2C master status mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyFlag

Tx shifter empty flag.

enumerator kFLEXIO_I2C_RxFullFlag

Rx shifter full/Transfer complete flag.

enumerator kFLEXIO_I2C_ReceiveNakFlag

Receive NAK flag.

enum _flexio_i2c_direction

Direction of master transfer.

Values:

enumerator kFLEXIO_I2C_Write

Master send to slave.

enumerator kFLEXIO_I2C_Read

Master receive from slave.

typedef enum _flexio_i2c_direction flexio_i2c_direction_t

Direction of master transfer.

typedef struct _flexio_i2c_type FLEXIO_I2C_Type

Define FlexIO I2C master access structure typedef.

typedef struct _flexio_i2c_master_config flexio_i2c_master_config_t

Define FlexIO I2C master user configuration structure.


```
typedef struct _flexio_i2c_master_transfer flexio_i2c_master_transfer_t
```

Define FlexIO I2C master transfer structure.

```
typedef struct _flexio_i2c_master_handle flexio_i2c_master_handle_t
```

FlexIO I2C master handle typedef.

```
typedef void (*flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type *base,  
flexio_i2c_master_handle_t *handle, status_t status, void *userData)
```

FlexIO I2C master transfer callback typedef.

```
I2C_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _flexio_i2c_type
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t SDAPinIndex

Pin select for I2C SDA.

uint8_t SCLPinIndex

Pin select for I2C SCL.

uint8_t shifterIndex[2]

Shifter index used in FlexIO I2C.

uint8_t timerIndex[3]

Timer index used in FlexIO I2C.

uint32_t baudrate

Master transfer baudrate, used to calculate delay time.

```
struct _flexio_i2c_master_config
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master user configuration structure.

Public Members

bool enableMaster

Enables the FlexIO I2C peripheral at initialization time.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

```
struct _flexio_i2c_master_transfer
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master transfer structure.

Public Members

uint32_t flags

Transfer flag which controls the transfer, reserved for FlexIO I2C.

uint8_t slaveAddress

7-bit slave address.

flexio_i2c_direction_t direction

Transfer direction, read or write.

uint32_t subaddress

Sub address. Transferred MSB first.

uint8_t subaddressSize

Size of sub address.

uint8_t volatile *data

Transfer buffer.

volatile size_t dataSize

Transfer size.

struct *_flexio_i2c_master_handle*

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master handle structure.

Public Members

flexio_i2c_master_transfer_t transfer

FlexIO I2C master transfer copy.

size_t transferSize

Total bytes to be transferred.

uint8_t state

Transfer state maintained during transfer.

flexio_i2c_master_transfer_callback_t completionCallback

Callback function called at transfer event. Callback function called at transfer event.

void *userData

Callback parameter passed to callback function.

bool needRestart

Whether master needs to send re-start signal.

2.25 FlexIO I2S Driver

void FLEXIO_I2S_Init(*FLEXIO_I2S_Type* *base, const *flexio_i2s_config_t* *config)

Initializes the FlexIO I2S.

This API configures FlexIO pins and shifter to I2S and configures the FlexIO I2S with a configuration structure. The configuration structure can be filled by the user, or be set with default values by FLEXIO_I2S_GetDefaultConfig().

Note: This API should be called at the beginning of the application to use the FlexIO I2S driver. Otherwise, any access to the FlexIO I2S module can cause hard fault because the clock is not enabled.

Parameters

- base – FlexIO I2S base pointer
- config – FlexIO I2S configure structure.

void FLEXIO_I2S_GetDefaultConfig(*flexio_i2s_config_t* *config)

Sets the FlexIO I2S configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in FLEXIO_I2S_Init(). Users may use the initialized structure unchanged in FLEXIO_I2S_Init() or modify some fields of the structure before calling FLEXIO_I2S_Init().

Parameters

- config – pointer to master configuration structure

void FLEXIO_I2S_Deinit(*FLEXIO_I2S_Type* *base)

De-initializes the FlexIO I2S.

Calling this API resets the FlexIO I2S shifter and timer config. After calling this API, call the FLEXIO_I2S_Init to use the FlexIO I2S module.

Parameters

- base – FlexIO I2S base pointer

static inline void FLEXIO_I2S_Enable(*FLEXIO_I2S_Type* *base, bool enable)

Enables/disables the FlexIO I2S module operation.

Parameters

- base – Pointer to FLEXIO_I2S_Type
- enable – True to enable, false dose not have any effect.

uint32_t FLEXIO_I2S_GetStatusFlags(*FLEXIO_I2S_Type* *base)

Gets the FlexIO I2S status flags.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure

Returns

Status flag, which are ORed by the enumerators in the `_flexio_i2s_status_flags`.

void FLEXIO_I2S_EnableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Enables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

void FLEXIO_I2S_DisableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Disables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

```
static inline void FLEXIO_I2S_TxEnableDMA(FLEXIO_I2S_Type *base, bool enable)
```

Enables/disables the FlexIO I2S Tx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

```
static inline void FLEXIO_I2S_RxEnableDMA(FLEXIO_I2S_Type *base, bool enable)
```

Enables/disables the FlexIO I2S Rx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

```
static inline uint32_t FLEXIO_I2S_TxGetDataRegisterAddress(FLEXIO_I2S_Type *base)
```

Gets the FlexIO I2S send data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure

Returns

FlexIO i2s send data register address.

```
static inline uint32_t FLEXIO_I2S_RxGetDataRegisterAddress(FLEXIO_I2S_Type *base)
```

Gets the FlexIO I2S receive data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure

Returns

FlexIO i2s receive data register address.

```
void FLEXIO_I2S_MasterSetFormat(FLEXIO_I2S_Type *base, flexio_i2s_format_t *format,  
                                uint32_t srcClock_Hz)
```

Configures the FlexIO I2S audio format in master mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- format – Pointer to FlexIO I2S audio data format structure.
- srcClock_Hz – I2S master clock source frequency in Hz.

```
void FLEXIO_I2S_SlaveSetFormat(FLEXIO_I2S_Type *base, flexio_i2s_format_t *format)
```

Configures the FlexIO I2S audio format in slave mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- format – Pointer to FlexIO I2S audio data format structure.

```
status_t FLEXIO_I2S_WriteBlocking(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint8_t *txData,  
                                size_t size)
```

Sends data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- txData – Pointer to the data to be written.
- size – Bytes to be written.

Return values

- kStatus_Success – Successfully write data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

```
static inline void FLEXIO_I2S_WriteData(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint32_t  
                                       data)
```

Writes data into a data register.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- data – Data to be written.

```
status_t FLEXIO_I2S_ReadBlocking(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint8_t *rxData,  
                                size_t size)
```

Receives a piece of data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- rxData – Pointer to the data to be read.
- size – Bytes to be read.

Return values

- kStatus_Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

```
static inline uint32_t FLEXIO_I2S_ReadData(FLEXIO_I2S_Type *base)
```

Reads a data from the data register.

Parameters

- base – FlexIO I2S base pointer

Returns

Data read from data register.

```
void FLEXIO_I2S_TransferTxCreateHandle(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                       flexio_i2s_callback_t callback, void *userData)
```

Initializes the FlexIO I2S handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- handle – Pointer to *flexio_i2s_handle_t* structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

```
void FLEXIO_I2S_TransferSetFormat(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                  flexio_i2s_format_t *format, uint32_t srcClock_Hz)
```

Configures the FlexIO I2S audio format.

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.
- handle – FlexIO I2S handle pointer.
- format – Pointer to audio data format structure.
- srcClock_Hz – FlexIO I2S bit clock source frequency in Hz. This parameter should be 0 while in slave mode.

```
void FLEXIO_I2S_TransferRxCreateHandle(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                       flexio_i2s_callback_t callback, void *userData)
```

Initializes the FlexIO I2S receive handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.
- handle – Pointer to *flexio_i2s_handle_t* structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

```
status_t FLEXIO_I2S_TransferSendNonBlocking(FLEXIO_I2S_Type *base, flexio_i2s_handle_t  
                                             *handle, flexio_i2s_transfer_t *xfer)
```

Performs an interrupt non-blocking send transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call *FLEXIO_I2S_GetRemainingBytes* to poll the transfer status and check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.

- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- xfer – Pointer to flexio_i2s_transfer_t structure

Return values

- kStatus_Success – Successfully start the data transmission.
- kStatus_FLEXIO_I2S_TxBusy – Previous transmission still not finished, data not all written to TX register yet.
- kStatus_InvalidArgument – The input parameter is invalid.

status_t FLEXIO_I2S_TransferReceiveNonBlocking(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_transfer_t* *xfer)

Performs an interrupt non-blocking receive transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call FLEXIO_I2S_GetRemainingBytes to poll the transfer status to check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- xfer – Pointer to flexio_i2s_transfer_t structure

Return values

- kStatus_Success – Successfully start the data receive.
- kStatus_FLEXIO_I2S_RxBusy – Previous receive still not finished.
- kStatus_InvalidArgument – The input parameter is invalid.

void FLEXIO_I2S_TransferAbortSend(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle)

Aborts the current send.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state

void FLEXIO_I2S_TransferAbortReceive(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle)

Aborts the current receive.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.

- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state

`status_t FLEXIO_I2S_TransferGetSendCount(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`

Gets the remaining bytes to be sent.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `count` – Bytes sent.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`status_t FLEXIO_I2S_TransferGetReceiveCount(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`

Gets the remaining bytes to be received.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `count` – Bytes recieved.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

Returns

`count` Bytes received.

`void FLEXIO_I2S_TransferTxHandleIRQ(void *i2sBase, void *i2sHandle)`

Tx interrupt handler.

Parameters

- `i2sBase` – Pointer to `FLEXIO_I2S_Type` structure.
- `i2sHandle` – Pointer to `flexio_i2s_handle_t` structure

`void FLEXIO_I2S_TransferRxHandleIRQ(void *i2sBase, void *i2sHandle)`

Rx interrupt handler.

Parameters

- `i2sBase` – Pointer to `FLEXIO_I2S_Type` structure.
- `i2sHandle` – Pointer to `flexio_i2s_handle_t` structure.

`FSL_FLEXIO_I2S_DRIVER_VERSION`

FlexIO I2S driver version 2.2.2.

FlexIO I2S transfer status.

Values:

enumerator kStatus_FLEXIO_I2S_Idle

FlexIO I2S is in idle state

enumerator kStatus_FLEXIO_I2S_TxBusy

FlexIO I2S Tx is busy

enumerator kStatus_FLEXIO_I2S_RxBusy

FlexIO I2S Rx is busy

enumerator kStatus_FLEXIO_I2S_Error

FlexIO I2S error occurred

enumerator kStatus_FLEXIO_I2S_QueueFull

FlexIO I2S transfer queue is full.

enumerator kStatus_FLEXIO_I2S_Timeout

FlexIO I2S timeout polling status flags.

enum _flexio_i2s_master_slave

Master or slave mode.

Values:

enumerator kFLEXIO_I2S_Master

Master mode

enumerator kFLEXIO_I2S_Slave

Slave mode

_flexio_i2s_interrupt_enable Define FlexIO FlexIO I2S interrupt mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyInterruptEnable

Transmit buffer empty interrupt enable.

enumerator kFLEXIO_I2S_RxDataRegFullInterruptEnable

Receive buffer full interrupt enable.

_flexio_i2s_status_flags Define FlexIO FlexIO I2S status mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyFlag

Transmit buffer empty flag.

enumerator kFLEXIO_I2S_RxDataRegFullFlag

Receive buffer full flag.

enum _flexio_i2s_sample_rate

Audio sample rate.

Values:

enumerator kFLEXIO_I2S_SampleRate8KHz

Sample rate 8000Hz

enumerator kFLEXIO_I2S_SampleRate11025Hz

Sample rate 11025Hz

enumerator kFLEXIO_I2S_SampleRate12KHz

Sample rate 12000Hz

enumerator kFLEXIO_I2S_SampleRate16KHz

Sample rate 16000Hz

enumerator kFLEXIO_I2S_SampleRate22050Hz

Sample rate 22050Hz

enumerator kFLEXIO_I2S_SampleRate24KHz

Sample rate 24000Hz

enumerator kFLEXIO_I2S_SampleRate32KHz

Sample rate 32000Hz

enumerator kFLEXIO_I2S_SampleRate44100Hz

Sample rate 44100Hz

enumerator kFLEXIO_I2S_SampleRate48KHz

Sample rate 48000Hz

enumerator kFLEXIO_I2S_SampleRate96KHz

Sample rate 96000Hz

enum _flexio_i2s_word_width

Audio word width.

Values:

enumerator kFLEXIO_I2S_WordWidth8bits

Audio data width 8 bits

enumerator kFLEXIO_I2S_WordWidth16bits

Audio data width 16 bits

enumerator kFLEXIO_I2S_WordWidth24bits

Audio data width 24 bits

enumerator kFLEXIO_I2S_WordWidth32bits

Audio data width 32 bits

typedef struct *_flexio_i2s_type* FLEXIO_I2S_Type

Define FlexIO I2S access structure typedef.

typedef enum *_flexio_i2s_master_slave* flexio_i2s_master_slave_t

Master or slave mode.

typedef struct *_flexio_i2s_config* flexio_i2s_config_t

FlexIO I2S configure structure.

typedef struct *_flexio_i2s_format* flexio_i2s_format_t

FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.

typedef enum *_flexio_i2s_sample_rate* flexio_i2s_sample_rate_t

Audio sample rate.

typedef enum *_flexio_i2s_word_width* flexio_i2s_word_width_t

Audio word width.

```
typedef struct _flexio_i2s_transfer flexio_i2s_transfer_t
    Define FlexIO I2S transfer structure.

typedef struct _flexio_i2s_handle flexio_i2s_handle_t

typedef void (*flexio_i2s_callback_t)(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,
status_t status, void *userData)
    FlexIO I2S xfer callback prototype.

I2S_RETRY_TIMES
    Retry times for waiting flag.

FLEXIO_I2S_XFER_QUEUE_SIZE
    FlexIO I2S transfer queue size, user can refine it according to use case.

struct _flexio_i2s_type
    #include <fsl_flexio_i2s.h> Define FlexIO I2S access structure typedef.
```

Public Members

```
FLEXIO_Type *flexioBase
    FlexIO base pointer

uint8_t txPinIndex
    Tx data pin index in FlexIO pins

uint8_t rxPinIndex
    Rx data pin index

uint8_t bclkPinIndex
    Bit clock pin index

uint8_t fsPinIndex
    Frame sync pin index

uint8_t txShifterIndex
    Tx data shifter index

uint8_t rxShifterIndex
    Rx data shifter index

uint8_t bclkTimerIndex
    Bit clock timer index

uint8_t fsTimerIndex
    Frame sync timer index

struct _flexio_i2s_config
    #include <fsl_flexio_i2s.h> FlexIO I2S configure structure.
```

Public Members

```
bool enableI2S
    Enable FlexIO I2S

flexio_i2s_master_slave_t masterSlave
    Master or slave

flexio_pin_polarity_t txPinPolarity
    Tx data pin polarity, active high or low
```

flexio_pin_polarity_t rxPinPolarity

Rx data pin polarity

flexio_pin_polarity_t bclkPinPolarity

Bit clock pin polarity

flexio_pin_polarity_t fsPinPolarity

Frame sync pin polarity

flexio_shifter_timer_polarity_t txTimerPolarity

Tx data valid on bclk rising or falling edge

flexio_shifter_timer_polarity_t rxTimerPolarity

Rx data valid on bclk rising or falling edge

struct *_flexio_i2s_format*

#include <fsl_flexio_i2s.h> FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.

Public Members

uint8_t bitWidth

Bit width of audio data, always 8/16/24/32 bits

uint32_t sampleRate_Hz

Sample rate of the audio data

struct *_flexio_i2s_transfer*

#include <fsl_flexio_i2s.h> Define FlexIO I2S transfer structure.

Public Members

uint8_t *data

Data buffer start pointer

size_t dataSize

Bytes to be transferred.

struct *_flexio_i2s_handle*

#include <fsl_flexio_i2s.h> Define FlexIO I2S handle structure.

Public Members

uint32_t state

Internal state

flexio_i2s_callback_t callback

Callback function called at transfer event

void *userData

Callback parameter passed to callback function

uint8_t bitWidth

Bit width for transfer, 8/16/24/32bits

flexio_i2s_transfer_t queue[(4U)]

Transfer queue storing queued transfer

```
size_t transferSize[(4U)]
```

Data bytes need to transfer

```
volatile uint8_t queueUser
```

Index for user to queue transfer

```
volatile uint8_t queueDriver
```

Index for driver to get the transfer data and size

2.26 FlexIO SPI Driver

```
void FLEXIO_SPI_MasterInit(FLEXIO_SPI_Type *base, flexio_spi_master_config_t
                           *masterConfig, uint32_t srcClock_Hz)
```

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI master hardware, and configures the FlexIO SPI with FlexIO SPI master configuration. The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_MasterGetDefaultConfig().

Example

```
FLEXIO_SPI_Type spiDev = {
    .flexioBase = FLEXIO,
    .SDOPinIndex = 0,
    .SDIPinIndex = 1,
    .SCKPinIndex = 2,
    .CSnPinIndex = 3,
    .shifterIndex = {0,1},
    .timerIndex = {0,1}
};
flexio_spi_master_config_t config = {
    .enableMaster = true,
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false,
    .baudRate_Bps = 500000,
    .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
    .direction = kFLEXIO_SPI_MsbFirst,
    .dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_MasterInit(&spiDev, &config, srcClock_Hz);
```

Note: 1.FlexIO SPI master only support CPOL = 0, which means clock inactive low. 2.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI master communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by $2*2=4$. If FlexIO SPI master communicates with FlexIO SPI slave, the maximum baud rate is FlexIO clock frequency divided by $(1.5+2.5)*2=8$.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- masterConfig – Pointer to the flexio_spi_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

```
void FLEXIO_SPI_MasterDeinit(FLEXIO_SPI_Type *base)
```

Resets the FlexIO SPI timer and shifter config.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type*.

```
void FLEXIO_SPI_MasterGetDefaultConfig(flexio_spi_master_config_t *masterConfig)
```

Gets the default configuration to configure the FlexIO SPI master. The configuration can be used directly by calling the *FLEXIO_SPI_MasterConfigure()*. Example:

```
flexio_spi_master_config_t masterConfig;  
FLEXIO_SPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – Pointer to the *flexio_spi_master_config_t* structure.

```
void FLEXIO_SPI_SlaveInit(FLEXIO_SPI_Type *base, flexio_spi_slave_config_t *slaveConfig)
```

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI slave hardware configuration, and configures the FlexIO SPI with FlexIO SPI slave configuration. The configuration structure can be filled by the user, or be set with default values by the *FLEXIO_SPI_SlaveGetDefaultConfig()*.

Note: 1. Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. 2. FlexIO SPI slave only support CPOL = 0, which means clock inactive low. 3. For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI slave communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by $3 \times 2 = 6$. If FlexIO SPI slave communicates with FlexIO SPI master, the maximum baud rate is FlexIO clock frequency divided by $(1.5 + 2.5) \times 2 = 8$. Example

```
FLEXIO_SPI_Type spiDev = {  
    .flexioBase = FLEXIO,  
    .SDOPinIndex = 0,  
    .SDIPinIndex = 1,  
    .SCKPinIndex = 2,  
    .CSnPinIndex = 3,  
    .shifterIndex = {0,1},  
    .timerIndex = {0}  
};  
flexio_spi_slave_config_t config = {  
    .enableSlave = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false,  
    .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,  
    .direction = kFLEXIO_SPI_MsbFirst,  
    .dataMode = kFLEXIO_SPI_8BitMode  
};  
FLEXIO_SPI_SlaveInit(&spiDev, &config);
```

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- slaveConfig – Pointer to the *flexio_spi_slave_config_t* structure.

void FLEXIO_SPI_SlaveDeinit(*FLEXIO_SPI_Type* *base)

Gates the FlexIO clock.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_SlaveGetDefaultConfig(*flexio_spi_slave_config_t* *slaveConfig)

Gets the default configuration to configure the FlexIO SPI slave. The configuration can be used directly for calling the FLEXIO_SPI_SlaveConfigure(). Example:

```
flexio_spi_slave_config_t slaveConfig;
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

uint32_t FLEXIO_SPI_GetStatusFlags(*FLEXIO_SPI_Type* *base)

Gets FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- kFLEXIO_SPI_TxEmptyFlag
- kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_ClearStatusFlags(*FLEXIO_SPI_Type* *base, uint32_t mask)

Clears FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – status flag The parameter can be any combination of the following values:
 - kFLEXIO_SPI_TxEmptyFlag
 - kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_EnableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Enables the FlexIO SPI interrupt.

This function enables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source. The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_DisableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Disables the FlexIO SPI interrupt.

This function disables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_EnableDMA(*FLEXIO_SPI_Type* *base, uint32_t mask, bool enable)

Enables/disables the FlexIO SPI transmit DMA. This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO_SPI_TxEmptyFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – SPI DMA source.
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t FLEXIO_SPI_GetTxDataRegisterAddress(*FLEXIO_SPI_Type* *base,
flexio_spi_shift_direction_t
direction)

Gets the FlexIO SPI transmit data register address for MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI transmit data register address.

static inline uint32_t FLEXIO_SPI_GetRxDataRegisterAddress(*FLEXIO_SPI_Type* *base,
flexio_spi_shift_direction_t
direction)

Gets the FlexIO SPI receive data register address for the MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI receive data register address.

static inline void FLEXIO_SPI_Enable(*FLEXIO_SPI_Type* *base, bool enable)

Enables/disables the FlexIO SPI module operation.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.
- enable – True to enable, false does not have any effect.

void FLEXIO_SPI_MasterSetBaudRate(*FLEXIO_SPI_Type* *base, uint32_t baudRate_Bps,
uint32_t srcClockHz)

Sets baud rate for the FlexIO SPI transfer, which is only used for the master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- baudRate_Bps – Baud Rate needed in Hz.
- srcClockHz – SPI source clock frequency in Hz.

```
static inline void FLEXIO_SPI_WriteData(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                         direction, uint32_t data)
```

Writes one byte of data, which is sent using the MSB method.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- data – 8/16/32 bit data.

```
static inline uint32_t FLEXIO_SPI_ReadData(FLEXIO_SPI_Type *base,  
                                           flexio_spi_shift_direction_t direction)
```

Reads 8 bit/16 bit data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

8 bit/16 bit data received.

```
status_t FLEXIO_SPI_WriteBlocking(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                   direction, const uint8_t *buffer, size_t size)
```

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The data bytes to send.
- size – The number of data bytes to send.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

status_t FLEXIO_SPI_ReadBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction, *uint8_t* *buffer, *size_t* size)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The buffer to store the received bytes.
- size – The number of data bytes to be received.

Return values

- *kStatus_Success* – Successfully create the handle.
- *kStatus_FLEXIO_SPI_Timeout* – The transfer timed out and was aborted.

status_t FLEXIO_SPI_MasterTransferBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_transfer_t* *xfer)

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – pointer to *FLEXIO_SPI_Type* structure
- xfer – FlexIO SPI transfer structure, see *flexio_spi_transfer_t*.

Return values

- *kStatus_Success* – Successfully create the handle.
- *kStatus_FLEXIO_SPI_Timeout* – The transfer timed out and was aborted.

void FLEXIO_SPI_FlushShifters(*FLEXIO_SPI_Type* *base)

Flush tx/rx shifters.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.

status_t FLEXIO_SPI_MasterTransferCreateHandle(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, *flexio_spi_master_transfer_callback_t* callback, *void* *userData)

Initializes the FlexIO SPI Master handle, which is used in transactional functions.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- handle – Pointer to the *flexio_spi_master_handle_t* structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- `kStatus_Success` – Successfully create the handle.
- `kStatus_OutOfRange` – The FlexIO type/handle/ISR table out of range.

`status_t` FLEXIO_SPI_MasterTransferNonBlocking(*FLEXIO_SPI_Type* *base,
 flexio_spi_master_handle_t *handle,
 flexio_spi_transfer_t *xfer)

Master transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.
- `xfer` – FlexIO SPI transfer structure. See `flexio_spi_transfer_t`.

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_FLEXIO_SPI_Busy` – SPI is not idle, is running another transfer.

`void` FLEXIO_SPI_MasterTransferAbort(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t*
 *handle)

Aborts the master data transfer, which used IRQ.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.

`status_t` FLEXIO_SPI_MasterTransferGetCount(*FLEXIO_SPI_Type* *base,
 flexio_spi_master_handle_t *handle, `size_t`
 *count)

Gets the data transfer status which used IRQ.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void` FLEXIO_SPI_MasterTransferHandleIRQ(`void` *spiType, `void` *spiHandle)

FlexIO SPI master IRQ handler function.

Parameters

- `spiType` – Pointer to the `FLEXIO_SPI_Type` structure.
- `spiHandle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.

```
status_t FLEXIO_SPI_SlaveTransferCreateHandle(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle,  
                                              flexio_spi_slave_transfer_callback_t callback,  
                                              void *userData)
```

Initializes the FlexIO SPI Slave handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

```
status_t FLEXIO_SPI_SlaveTransferNonBlocking(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle,  
                                              flexio_spi_transfer_t *xfer)
```

Slave transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- base – Pointer to the FLEXIO_SPI_Type structure.
- xfer – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – SPI is not idle; it is running another transfer.

```
static inline void FLEXIO_SPI_SlaveTransferAbort(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle)
```

Aborts the slave data transfer which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

```
static inline status_t FLEXIO_SPI_SlaveTransferGetCount(FLEXIO_SPI_Type *base,  
                                                       flexio_spi_slave_handle_t *handle,  
                                                       size_t *count)
```

Gets the data transfer status which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void FLEXIO_SPI_SlaveTransferHandleIRQ(void *spiType, void *spiHandle)`

FlexIO SPI slave IRQ handler function.

Parameters

- `spiType` – Pointer to the `FLEXIO_SPI_Type` structure.
- `spiHandle` – Pointer to the `flexio_spi_slave_handle_t` structure to store the transfer state.

`FSL_FLEXIO_SPI_DRIVER_VERSION`

FlexIO SPI driver version.

Error codes for the FlexIO SPI driver.

Values:

enumerator `kStatus_FLEXIO_SPI_Busy`
FlexIO SPI is busy.

enumerator `kStatus_FLEXIO_SPI_Idle`
SPI is idle

enumerator `kStatus_FLEXIO_SPI_Error`
FlexIO SPI error.

enumerator `kStatus_FLEXIO_SPI_Timeout`
FlexIO SPI timeout polling status flags.

enum `_flexio_spi_clock_phase`

FlexIO SPI clock phase configuration.

Values:

enumerator `kFLEXIO_SPI_ClockPhaseFirstEdge`
First edge on SPSCCK occurs at the middle of the first cycle of a data transfer.

enumerator `kFLEXIO_SPI_ClockPhaseSecondEdge`
First edge on SPSCCK occurs at the start of the first cycle of a data transfer.

enum `_flexio_spi_shift_direction`

FlexIO SPI data shifter direction options.

Values:

enumerator `kFLEXIO_SPI_MsbFirst`
Data transfers start with most significant bit.

enumerator `kFLEXIO_SPI_LsbFirst`
Data transfers start with least significant bit.

enum `_flexio_spi_data_bitcount_mode`

FlexIO SPI data length mode options.

Values:

enumerator `kFLEXIO_SPI_8BitMode`
8-bit data transmission mode.

enumerator kFLEXIO_SPI_16BitMode
16-bit data transmission mode.

enumerator kFLEXIO_SPI_32BitMode
32-bit data transmission mode.

enum _flexio_spi_interrupt_enable
Define FlexIO SPI interrupt mask.

Values:

enumerator kFLEXIO_SPI_TxEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_SPI_RxFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_spi_status_flags
Define FlexIO SPI status mask.

Values:

enumerator kFLEXIO_SPI_TxBufferEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_SPI_RxBufferFullFlag
Receive buffer full flag.

enum _flexio_spi_dma_enable
Define FlexIO SPI DMA mask.

Values:

enumerator kFLEXIO_SPI_TxDmaEnable
Tx DMA request source

enumerator kFLEXIO_SPI_RxDmaEnable
Rx DMA request source

enumerator kFLEXIO_SPI_DmaAllEnable
All DMA request source

enum _flexio_spi_transfer_flags
Define FlexIO SPI transfer flags.

Note: Use kFLEXIO_SPI_csContinuous and one of the other flags to OR together to form the transfer flag.

Values:

enumerator kFLEXIO_SPI_8bitMsb
FlexIO SPI 8-bit MSB first

enumerator kFLEXIO_SPI_8bitLsb
FlexIO SPI 8-bit LSB first

enumerator kFLEXIO_SPI_16bitMsb
FlexIO SPI 16-bit MSB first

enumerator kFLEXIO_SPI_16bitLsb
FlexIO SPI 16-bit LSB first

```

enumerator kFLEXIO__SPI__32bitMsb
    FlexIO SPI 32-bit MSB first
enumerator kFLEXIO__SPI__32bitLsb
    FlexIO SPI 32-bit LSB first
enumerator kFLEXIO__SPI__csContinuous
    Enable the CS signal continuous mode
typedef enum _flexio_spi_clock_phase flexio_spi_clock_phase_t
    FlexIO SPI clock phase configuration.
typedef enum _flexio_spi_shift_direction flexio_spi_shift_direction_t
    FlexIO SPI data shifter direction options.
typedef enum _flexio_spi_data_bitcount_mode flexio_spi_data_bitcount_mode_t
    FlexIO SPI data length mode options.
typedef struct _flexio_spi_type FLEXIO__SPI_Type
    Define FlexIO SPI access structure typedef.
typedef struct _flexio_spi_master_config flexio_spi_master_config_t
    Define FlexIO SPI master configuration structure.
typedef struct _flexio_spi_slave_config flexio_spi_slave_config_t
    Define FlexIO SPI slave configuration structure.
typedef struct _flexio_spi_transfer flexio_spi_transfer_t
    Define FlexIO SPI transfer structure.
typedef struct _flexio_spi_master_handle flexio_spi_master_handle_t
    typedef for flexio_spi_master_handle_t in advance.
typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t
    Slave handle is the same with master handle.
typedef void (*flexio_spi_master_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_master_handle_t *handle, status_t status, void *userData)
    FlexIO SPI master callback for finished transmit.
typedef void (*flexio_spi_slave_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_slave_handle_t *handle, status_t status, void *userData)
    FlexIO SPI slave callback for finished transmit.
FLEXIO__SPI__DUMMYDATA
    FlexIO SPI dummy transfer data, the data is sent while txData is NULL.
SPI_RETRY_TIMES
    Retry times for waiting flag.
FLEXIO_SPI_XFER_DATA_FORMAT(flag)
    Get the transfer data format of width and bit order.
struct _flexio_spi_type
    #include <fsl_flexio_spi.h> Define FlexIO SPI access structure typedef.

```

Public Members

```

FLEXIO_Type *flexioBase
    FlexIO base pointer.

```

uint8_t SDOPinIndex

Pin select for data output. To set SDO pin in Hi-Z state, user needs to mux the pin as GPIO input and disable all pull up/down in application.

uint8_t SDIPinIndex

Pin select for data input.

uint8_t SCKPinIndex

Pin select for clock.

uint8_t CSnPinIndex

Pin select for enable.

uint8_t shifterIndex[2]

Shifter index used in FlexIO SPI.

uint8_t timerIndex[2]

Timer index used in FlexIO SPI.

struct __flexio_spi_master_config

#include <fsl_flexio_spi.h> Define FlexIO SPI master configuration structure.

Public Members

bool enableMaster

Enable/disable FlexIO SPI master after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct __flexio_spi_slave_config

#include <fsl_flexio_spi.h> Define FlexIO SPI slave configuration structure.

Public Members

bool enableSlave

Enable/disable FlexIO SPI slave after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct *_flexio_spi_transfer*

#include <fsl_flexio_spi.h> Define FlexIO SPI transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

size_t dataSize

Transfer bytes.

uint8_t flags

FlexIO SPI control flag, MSB first or LSB first.

struct *_flexio_spi_master_handle*

#include <fsl_flexio_spi.h> Define FlexIO SPI handle structure.

Public Members

const uint8_t *txData

Transfer buffer.

uint8_t *rxData

Receive buffer.

size_t transferSize

Total bytes to be transferred.

volatile size_t txRemainingBytes

Send data remaining in bytes.

volatile size_t rxRemainingBytes

Receive data remaining in bytes.

volatile uint32_t state

FlexIO SPI internal state.

uint8_t bytePerFrame

SPI mode, 2bytes or 1byte in a frame

flexio_spi_shift_direction_t direction

Shift direction.

flexio_spi_master_transfer_callback_t callback

FlexIO SPI callback.

void *userData

Callback parameter.

bool isCsContinuous

Is current transfer using CS continuous mode.

uint32_t timer1Cfg

TIMER1 TIMCFG register value backup.

2.27 FlexIO UART Driver

status_t FLEXIO_UART_Init(*FLEXIO_UART_Type* *base, const *flexio_uart_config_t* *userConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, configures FlexIO UART hardware, and configures the FlexIO UART with FlexIO UART configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_UART_GetDefaultConfig().

Example

```
FLEXIO_UART_Type base = {
    .flexioBase = FLEXIO,
    .TxPinIndex = 0,
    .RxPinIndex = 1,
    .shifterIndex = {0,1},
    .timerIndex = {0,1}
};
flexio_uart_config_t config = {
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false,
    .baudRate_Bps = 115200U,
    .bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- userConfig – Pointer to the flexio_uart_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Configuration success.
- kStatus_FLEXIO_UART_BaudrateNotSupport – Baudrate is not supported for current clock source frequency.

void FLEXIO_UART_Deinit(*FLEXIO_UART_Type* *base)

Resets the FlexIO UART shifter and timer config.

Note: After calling this API, call the FLEXIO_UART_Init to use the FlexIO UART module.

Parameters

- base – Pointer to FLEXIO_UART_Type structure

void FLEXIO_UART_GetDefaultConfig(*flexio_uart_config_t* *userConfig)

Gets the default configuration to configure the FlexIO UART. The configuration can be used directly for calling the FLEXIO_UART_Init(). Example:

```
flexio_uart_config_t config;
FLEXIO_UART_GetDefaultConfig(&userConfig);
```

Parameters

- userConfig – Pointer to the flexio_uart_config_t structure.

uint32_t FLEXIO_UART_GetStatusFlags(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART status flags.

void FLEXIO_UART_ClearStatusFlags(*FLEXIO_UART_Type* *base, uint32_t mask)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
 - kFLEXIO_UART_TxDataRegEmptyFlag
 - kFLEXIO_UART_RxEmptyFlag
 - kFLEXIO_UART_RxOverRunFlag

void FLEXIO_UART_EnableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Enables the FlexIO UART interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

void FLEXIO_UART_DisableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Disables the FlexIO UART interrupt.

This function disables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

static inline uint32_t FLEXIO_UART_GetTxDataRegisterAddress(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART transmit data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART transmit data register address.

```
static inline uint32_t FLEXIO_UART_GetRxDataRegisterAddress(FLEXIO_UART_Type *base)
```

Gets the FlexIO UART receive data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.

Returns

FlexIO UART receive data register address.

```
static inline void FLEXIO_UART_EnableTxDMA(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART transmit DMA. This function enables/disables the FlexIO UART Tx DMA, which means asserting the *kFLEXIO_UART_TxDataRegEmptyFlag* does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- enable – True to enable, false to disable.

```
static inline void FLEXIO_UART_EnableRxDMA(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART receive DMA. This function enables/disables the FlexIO UART Rx DMA, which means asserting *kFLEXIO_UART_RxDataRegFullFlag* does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- enable – True to enable, false to disable.

```
static inline void FLEXIO_UART_Enable(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART module operation.

Parameters

- base – Pointer to the *FLEXIO_UART_Type*.
- enable – True to enable, false does not have any effect.

```
static inline void FLEXIO_UART_WriteByte(FLEXIO_UART_Type *base, const uint8_t *buffer)
```

Writes one byte of data.

Note: This is a non-blocking API, which returns directly after the data is put into the data register. Ensure that the *TxEmptyFlag* is asserted before calling this API.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- buffer – The data bytes to send.

```
static inline void FLEXIO_UART_ReadByte(FLEXIO_UART_Type *base, uint8_t *buffer)
```

Reads one byte of data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the *RxFullFlag* is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- buffer – The buffer to store the received bytes.

status_t FLEXIO_UART_WriteBlocking(*FLEXIO_UART_Type* *base, const uint8_t *txData, size_t txSize)

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- txData – The data bytes to send.
- txSize – The number of data bytes to send.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t FLEXIO_UART_ReadBlocking(*FLEXIO_UART_Type* *base, uint8_t *rxData, size_t rxSize)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- rxData – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t FLEXIO_UART_TransferCreateHandle(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *flexio_uart_transfer_callback_t* callback, void *userData)

Initializes the UART handle.

This function initializes the FlexIO UART handle, which can be used for other FlexIO UART transactional APIs. Call this API once to get the initialized handle.

The UART driver supports the “background” receiving, which means that users can set up a RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn’t call the FLEXIO_UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as ringBuffer.

Parameters

- base – to FLEXIO_UART_Type structure.

- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `callback` – The callback function.
- `userData` – The parameter of the callback function.

Return values

- `kStatus_Success` – Successfully create the handle.
- `kStatus_OutOfRange` – The FlexIO type/handle/ISR table out of range.

```
void FLEXIO_UART_TransferStartRingBuffer(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                         *handle, uint8_t *ringBuffer, size_t
                                         ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the `UART_ReceiveNonBlocking()` API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `ringBuffer` – Start address of ring buffer for background receiving. Pass `NULL` to disable the ring buffer.
- `ringBufferSize` – Size of the ring buffer.

```
void FLEXIO_UART_TransferStopRingBuffer(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                         *handle)
```

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.

```
status_t FLEXIO_UART_TransferSendNonBlocking(FLEXIO_UART_Type *base,
                                              flexio_uart_handle_t *handle,
                                              flexio_uart_transfer_t *xfer)
```

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in ISR, the FlexIO UART driver calls the callback function and passes the `kStatus_FLEXIO_UART_TxIdle` as status parameter.

Note: The `kStatus_FLEXIO_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `xfer` – FlexIO UART transfer structure. See `flexio_uart_transfer_t`.

Return values

- `kStatus_Success` – Successfully starts the data transmission.
- `kStatus_UART_TxBusy` – Previous transmission still not finished, data not written to the TX register.

```
void FLEXIO_UART_TransferAbortSend(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                   *handle)
```

Aborts the interrupt-driven data transmit.

This function aborts the interrupt-driven data sending. Get the `remainBytes` to find out how many bytes are still not sent out.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.

```
status_t FLEXIO_UART_TransferGetSendCount(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                           *handle, size_t *count)
```

Gets the number of bytes sent.

This function gets the number of bytes sent driven by interrupt.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `count` – Number of bytes sent so far by the non-blocking transaction.

Return values

- `kStatus_NoTransferInProgress` – transfer has finished or no transfer in progress.
- `kStatus_Success` – Successfully return the count.

```
status_t FLEXIO_UART_TransferReceiveNonBlocking(FLEXIO_UART_Type *base,
                                                flexio_uart_handle_t *handle,
                                                flexio_uart_transfer_t *xfer, size_t
                                                *receivedBytes)
```

Receives a buffer of data using the interrupt method.

This function receives data using the interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in ring buffer is not enough to read, the receive request is saved by the UART driver. When new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_UART_RxIdle`. For example, if the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer, the 5 bytes are copied to `xfer->data`. This function returns with the parameter `receivedBytes` set to 5. For the last 5 bytes, newly arrived data is saved from the

xfer->data[5]. When 5 bytes are received, the UART driver notifies upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to xfer->data. When all data is received, the upper layer is notified.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- xfer – UART transfer structure. See flexio_uart_transfer_t.
- receivedBytes – Bytes received from the ring buffer directly.

Return values

- kStatus__Success – Successfully queue the transfer into the transmit queue.
- kStatus_FLEXIO_UART_RxBusy – Previous receive request is not finished.

void FLEXIO_UART_TransferAbortReceive(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle)

Aborts the receive data which was using IRQ.

This function aborts the receive data which was using IRQ.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

status_t FLEXIO_UART_TransferGetReceiveCount(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *size_t* *count)

Gets the number of bytes received.

This function gets the number of bytes received driven by interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- count – Number of bytes received so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus__Success – Successfully return the count.

void FLEXIO_UART_TransferHandleIRQ(void *uartType, void *uartHandle)

FlexIO UART IRQ handler function.

This function processes the FlexIO UART transmit and receives the IRQ request.

Parameters

- uartType – Pointer to the FLEXIO_UART_Type structure.
- uartHandle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

void FLEXIO_UART_FlushShifters(*FLEXIO_UART_Type* *base)
 Flush tx/rx shifters.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

FSL_FLEXIO_UART_DRIVER_VERSION
 FlexIO UART driver version.

Error codes for the UART driver.

Values:

enumerator kStatus_FLEXIO_UART_TxBusy
 Transmitter is busy.

enumerator kStatus_FLEXIO_UART_RxBusy
 Receiver is busy.

enumerator kStatus_FLEXIO_UART_TxIdle
 UART transmitter is idle.

enumerator kStatus_FLEXIO_UART_RxIdle
 UART receiver is idle.

enumerator kStatus_FLEXIO_UART_ERROR
 ERROR happens on UART.

enumerator kStatus_FLEXIO_UART_RxRingBufferOverrun
 UART RX software ring buffer overrun.

enumerator kStatus_FLEXIO_UART_RxHardwareOverrun
 UART RX receiver overrun.

enumerator kStatus_FLEXIO_UART_Timeout
 UART times out.

enumerator kStatus_FLEXIO_UART_BaudrateNotSupport
 Baudrate is not supported in current clock source

enum _flexio_uart_bit_count_per_char
 FlexIO UART bit count per char.

Values:

enumerator kFLEXIO_UART_7BitsPerChar
 7-bit data characters

enumerator kFLEXIO_UART_8BitsPerChar
 8-bit data characters

enumerator kFLEXIO_UART_9BitsPerChar
 9-bit data characters

enum _flexio_uart_interrupt_enable
 Define FlexIO UART interrupt mask.

Values:

enumerator kFLEXIO_UART_TxDataRegEmptyInterruptEnable
 Transmit buffer empty interrupt enable.

enumerator kFLEXIO_UART_RxDataRegFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_uart_status_flags
Define FlexIO UART status mask.

Values:

enumerator kFLEXIO_UART_TxDataRegEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_UART_RxDataRegFullFlag
Receive buffer full flag.

enumerator kFLEXIO_UART_RxOverRunFlag
Receive buffer over run flag.

typedef enum _flexio_uart_bit_count_per_char flexio_uart_bit_count_per_char_t
FlexIO UART bit count per char.

typedef struct _flexio_uart_type FLEXIO_UART_Type
Define FlexIO UART access structure typedef.

typedef struct _flexio_uart_config flexio_uart_config_t
Define FlexIO UART user configuration structure.

typedef struct _flexio_uart_transfer flexio_uart_transfer_t
Define FlexIO UART transfer structure.

typedef struct _flexio_uart_handle flexio_uart_handle_t

typedef void (*flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)
FlexIO UART transfer callback function.

UART_RETRY_TIMES
Retry times for waiting flag.

struct _flexio_uart_type
#include <fsl_flexio_uart.h> Define FlexIO UART access structure typedef.

Public Members

FLEXIO_Type *flexioBase
FlexIO base pointer.

uint8_t TxPinIndex
Pin select for UART_Tx.

uint8_t RxPinIndex
Pin select for UART_Rx.

uint8_t shifterIndex[2]
Shifter index used in FlexIO UART.

uint8_t timerIndex[2]
Timer index used in FlexIO UART.

struct _flexio_uart_config
#include <fsl_flexio_uart.h> Define FlexIO UART user configuration structure.

Public Members

`bool enableUart`
Enable/disable FlexIO UART TX & RX.

`bool enableInDoze`
Enable/disable FlexIO operation in doze mode

`bool enableInDebug`
Enable/disable FlexIO operation in debug mode

`bool enableFastAccess`
Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

`uint32_t baudRate_Bps`
Baud rate in Bps.

`flexio_uart_bit_count_per_char_t bitCountPerChar`
number of bits, 7/8/9 -bit

`struct _flexio_uart_transfer`
#include <fsl_flexio_uart.h> Define FlexIO UART transfer structure.

Public Members

`size_t dataSize`
Transfer size

`struct _flexio_uart_handle`
#include <fsl_flexio_uart.h> Define FLEXIO UART handle structure.

Public Members

`const uint8_t *volatile txData`
Address of remaining data to send.

`volatile size_t txDataSize`
Size of the remaining data to send.

`uint8_t *volatile rxData`
Address of remaining data to receive.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t txDataSizeAll`
Total bytes to be sent.

`size_t rxDataSizeAll`
Total bytes to be received.

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
Index for the user to get data from the ring buffer.

flexio_uart_transfer_callback_t callback
Callback function.

void *userData
UART callback function parameter.

volatile uint8_t txState
TX transfer state.

volatile uint8_t rxState
RX transfer state

union __unnamed78__

Public Members

uint8_t *data
The buffer of data to be transfer.

uint8_t *rxData
The buffer to receive data.

const uint8_t *txData
The buffer of data to be sent.

2.28 GPIO: General-Purpose Input/Output Driver

FSL_GPIO_DRIVER_VERSION
GPIO driver version.

enum _gpio_pin_direction
GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput
Set current pin as digital input

enumerator kGPIO_DigitalOutput
Set current pin as digital output

enum _gpio_checker_attribute
GPIO checker attribute.

Values:

enumerator kGPIO_UsernonsecureRWUsersecureRWPrivilegedsecureRW
User nonsecure:Read+Write; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureRUsersecureRWPrivilegedsecureRW
User nonsecure:Read; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRWPrivilegedsecureRW
User nonsecure:None; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO__UsernonsecureRUsersecureRPrivilegedsecureRW
 User nonsecure:Read; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO__UsernonsecureNUsersecureRPrivilegedsecureRW
 User nonsecure:None; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO__UsernonsecureNUsersecureNPrivilegedsecureRW
 User nonsecure:None; User Secure:None; Privileged Secure:Read+Write

enumerator kGPIO__UsernonsecureNUsersecureNPrivilegedsecureR
 User nonsecure:None; User Secure:None; Privileged Secure:Read

enumerator kGPIO__UsernonsecureNUsersecureNPrivilegedsecureN
 User nonsecure:None; User Secure:None; Privileged Secure:None

enumerator kGPIO__IgnoreAttributeCheck
 Ignores the attribute check

enum __gpio_interrupt_config

Configures the interrupt generation condition.

Values:

enumerator kGPIO__InterruptStatusFlagDisabled
 Interrupt status flag is disabled.

enumerator kGPIO__DMARisingEdge
 ISF flag and DMA request on rising edge.

enumerator kGPIO__DMAFallingEdge
 ISF flag and DMA request on falling edge.

enumerator kGPIO__DMAEitherEdge
 ISF flag and DMA request on either edge.

enumerator kGPIO__FlagRisingEdge
 Flag sets on rising edge.

enumerator kGPIO__FlagFallingEdge
 Flag sets on falling edge.

enumerator kGPIO__FlagEitherEdge
 Flag sets on either edge.

enumerator kGPIO__InterruptLogicZero
 Interrupt when logic zero.

enumerator kGPIO__InterruptRisingEdge
 Interrupt on rising edge.

enumerator kGPIO__InterruptFallingEdge
 Interrupt on falling edge.

enumerator kGPIO__InterruptEitherEdge
 Interrupt on either edge.

enumerator kGPIO__InterruptLogicOne
 Interrupt when logic one.

enumerator kGPIO__ActiveHighTriggerOutputEnable
 Enable active high-trigger output.

enumerator kGPIO__ActiveLowTriggerOutputEnable

Enable active low-trigger output.

enum __gpio_interrupt_selection

Configures the selection of interrupt/DMA request/trigger output.

Values:

enumerator kGPIO__InterruptOutput0

Interrupt/DMA request/trigger output 0.

enumerator kGPIO__InterruptOutput1

Interrupt/DMA request/trigger output 1.

enum gpio_pin_interrupt_control_t

GPIO pin and interrupt control.

Values:

enumerator kGPIO__PinControlNonSecure

Pin Control Non-Secure.

enumerator kGPIO__InterruptControlNonSecure

Interrupt Control Non-Secure.

enumerator kGPIO__PinControlNonPrivilege

Pin Control Non-Privilege.

enumerator kGPIO__InterruptControlNonPrivilege

Interrupt Control Non-Privilege.

typedef enum __gpio_pin_direction gpio_pin_direction_t

GPIO direction definition.

typedef enum __gpio_checker_attribute gpio_checker_attribute_t

GPIO checker attribute.

typedef struct __gpio_pin_config gpio_pin_config_t

The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

typedef enum __gpio_interrupt_config gpio_interrupt_config_t

Configures the interrupt generation condition.

typedef enum __gpio_interrupt_selection gpio_interrupt_selection_t

Configures the selection of interrupt/DMA request/trigger output.

typedef struct __gpio_version_info gpio_version_info_t

GPIO version information.

GPIO__FIT__REG(value)

struct __gpio_pin_config

#include <fsl_gpio.h> The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

Public Members*gpio_pin_direction_t* pinDirection

GPIO direction, input or output

uint8_t outputLogic

Set a default output logic, which has no use in input

struct __gpio_version_info

#include <fsl_gpio.h> GPIO version information.

Public Members

uint16_t feature

Feature Specification Number.

uint8_t minor

Minor Version Number.

uint8_t major

Major Version Number.

2.29 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)

Deinitializes the GPIO peripheral.

Parameters

- base – GPIO peripheral base pointer.

void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const *gpio_pin_config_t* *config)

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or an output pin configuration.

```

Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}

```

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO port pin number
- config – GPIO pin configuration pointer

```
void GPIO_GetVersionInfo(GPIO_Type *base, gpio_version_info_t *verInfo)
```

Get GPIO version information.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- verInfo – GPIO version information

```
static inline void GPIO_SecurePrivilegeLock(GPIO_Type *base, gpio_pin_interrupt_control_t  
mask)
```

lock or unlock secure privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – pin or interrupt macro

```
static inline void GPIO_EnablePinControlNonSecure(GPIO_Type *base, uint32_t mask)
```

Enable Pin Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_DisablePinControlNonSecure(GPIO_Type *base, uint32_t mask)
```

Disable Pin Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_EnablePinControlNonPrivilege(GPIO_Type *base, uint32_t mask)
```

Enable Pin Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_DisablePinControlNonPrivilege(GPIO_Type *base, uint32_t mask)
```

Disable Pin Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_EnableInterruptControlNonSecure(GPIO_Type *base, uint32_t mask)
```

Enable Interrupt Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisableInterruptControlNonSecure(GPIO_Type *base, uint32_t mask)
Disable Interrupt Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_EnableInterruptControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Enable Interrupt Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisableInterruptControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Disable Interrupt Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortInputEnable(GPIO_Type *base, uint32_t mask)
Enable port input.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortInputDisable(GPIO_Type *base, uint32_t mask)
Disable port input.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t pin, uint8_t output)
Sets the output level of the multiple GPIO pins to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t mask)
Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_PortClear(GPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t mask)
```

Reverses the current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

```
static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t pin)
```

Reads the current input value of the GPIO port.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

```
static inline void GPIO_SetPinInterruptConfig(GPIO_Type *base, uint32_t pin,  
                                              gpio_interrupt_config_t config)
```

Configures the gpio pin interrupt/DMA request.

Parameters

- base – GPIO peripheral base pointer.
- pin – GPIO pin number.
- config – GPIO pin interrupt configuration.
 - kGPIO_InterruptStatusFlagDisabled: Interrupt/DMA request disabled.
 - kGPIO_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
 - kGPIO_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
 - kGPIO_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
 - kGPIO_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
 - kGPIO_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
 - kGPIO_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
 - kGPIO_InterruptLogicZero : Interrupt when logic zero.
 - kGPIO_InterruptRisingEdge : Interrupt on rising edge.
 - kGPIO_InterruptFallingEdge: Interrupt on falling edge.
 - kGPIO_InterruptEitherEdge : Interrupt on either edge.
 - kGPIO_InterruptLogicOne : Interrupt when logic one.

- kGPIO_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
- kGPIO_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

```
static inline void GPIO_SetPinInterruptChannel(GPIO_Type *base, uint32_t pin,
                                             gpio_interrupt_selection_t selection)
```

Configures the gpio pin interrupt/DMA request/trigger output channel selection.

Parameters

- base – GPIO peripheral base pointer.
- pin – GPIO pin number.
- selection – GPIO pin interrupt output selection.
 - kGPIO_InterruptOutput0: Interrupt/DMA request/trigger output 0.
 - kGPIO_InterruptOutput1 : Interrupt/DMA request/trigger output 1.

```
uint32_t GPIO_GpioGetInterruptFlags(GPIO_Type *base)
```

Read the GPIO interrupt status flags.

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on.)

Returns

The current GPIO's interrupt status flag. '1' means the related pin's flag is set, '0' means the related pin's flag not set. For example, the return value 0x00010001 means the pin 0 and 17 have the interrupt pending.

```
uint32_t GPIO_GpioGetInterruptChannelFlags(GPIO_Type *base, uint32_t channel)
```

Read the GPIO interrupt status flags based on selected interrupt channel(IRQS).

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on.)
- channel – '0' means selete interrupt channel 0, '1' means selete interrupt channel 1.

Returns

The current GPIO's interrupt status flag based on the selected interrupt channel. '1' means the related pin's flag is set, '0' means the related pin's flag not set. For example, the return value 0x00010001 means the pin 0 and 17 have the interrupt pending.

```
uint8_t GPIO_PinGetInterruptFlag(GPIO_Type *base, uint32_t pin)
```

Read individual pin's interrupt status flag.

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on)
- pin – GPIO specific pin number.

Returns

The current selected pin's interrupt status flag.

```
void GPIO_GpioClearInterruptFlags(GPIO_Type *base, uint32_t mask)
```

Clears GPIO pin interrupt status flags.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

`void GPIO_GpioClearInterruptChannelFlags(GPIO_Type *base, uint32_t mask, uint32_t channel)`
Clears GPIO pin interrupt status flags based on selected interrupt channel(IRQS).

Parameters

- `base` – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- `mask` – GPIO pin number macro
- `channel` – ‘0’ means selete interrupt channel 0, ‘1’ means selete interrupt channel 1.

`void GPIO_PinClearInterruptFlag(GPIO_Type *base, uint32_t pin)`
Clear GPIO individual pin’s interrupt status flag.

Parameters

- `base` – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on).
- `pin` – GPIO specific pin number.

`static inline void GPIO_SetMultipleInterruptPinsConfig(GPIO_Type *base, uint32_t mask, gpio_interrupt_config_t config)`

Sets the GPIO interrupt configuration in PCR register for multiple pins.

Parameters

- `base` – GPIO peripheral base pointer.
- `mask` – GPIO pin number macro.
- `config` – GPIO pin interrupt configuration.
 - `kGPIO_InterruptStatusFlagDisabled`: Interrupt disabled.
 - `kGPIO_DMARisingEdge` : DMA request on rising edge(if the DMA requests exit).
 - `kGPIO_DMAFallingEdge`: DMA request on falling edge(if the DMA requests exit).
 - `kGPIO_DMAEitherEdge` : DMA request on either edge(if the DMA requests exit).
 - `kGPIO_FlagRisingEdge` : Flag sets on rising edge(if the Flag states exit).
 - `kGPIO_FlagFallingEdge` : Flag sets on falling edge(if the Flag states exit).
 - `kGPIO_FlagEitherEdge` : Flag sets on either edge(if the Flag states exit).
 - `kGPIO_InterruptLogicZero` : Interrupt when logic zero.
 - `kGPIO_InterruptRisingEdge` : Interrupt on rising edge.
 - `kGPIO_InterruptFallingEdge`: Interrupt on falling edge.
 - `kGPIO_InterruptEitherEdge` : Interrupt on either edge.
 - `kGPIO_InterruptLogicOne` : Interrupt when logic one.
 - `kGPIO_ActiveHighTriggerOutputEnable` : Enable active high-trigger output (if the trigger states exit).
 - `kGPIO_ActiveLowTriggerOutputEnable` : Enable active low-trigger output (if the trigger states exit)..

`void GPIO_CheckAttributeBytes(GPIO_Type *base, gpio_checker_attribute_t attribute)`

brief The GPIO module supports a device-specific number of data ports, organized as 32-bit words/8-bit Bytes. Each 32-bit/8-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the GPIO programming model. If

the GPIO module's GACR register organized as 32-bit words, the attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- attribute – GPIO checker attribute

2.30 I3C: I3C Driver

FSL_I3C_DRIVER_VERSION

I3C driver version.

I3C status return codes.

Values:

enumerator kStatus_I3C_Busy

The master is already performing a transfer.

enumerator kStatus_I3C_Idle

The slave driver is idle.

enumerator kStatus_I3C_Nak

The slave device sent a NAK in response to an address.

enumerator kStatus_I3C_WriteAbort

The slave device sent a NAK in response to a write.

enumerator kStatus_I3C_Term

The master terminates slave read.

enumerator kStatus_I3C_HdrParityError

Parity error from DDR read.

enumerator kStatus_I3C_CrcError

CRC error from DDR read.

enumerator kStatus_I3C_ReadFifoError

Read from M/SRDATAB register when FIFO empty.

enumerator kStatus_I3C_WriteFifoError

Write to M/SWDATAB register when FIFO full.

enumerator kStatus_I3C_MsgError

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kStatus_I3C_InvalidReq

Invalid use of request.

enumerator kStatus_I3C_Timeout

The module has stalled too long in a frame.

enumerator kStatus_I3C_SlaveCountExceed

The I3C slave count has exceed the definition in I3C_MAX_DEVCNT.

enumerator kStatus_I3C_IBIWon

The I3C slave event IBI or MR or HJ won the arbitration on a header address.

enumerator kStatus_I3C_OverrunError

Slave internal from-bus buffer/FIFO overrun.

enumerator kStatus_I3C_UnderrunError

Slave internal to-bus buffer/FIFO underrun

enumerator kStatus_I3C_UnderrunNak

Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kStatus_I3C_InvalidStart

Slave invalid start flag

enumerator kStatus_I3C_SdrParityError

SDR parity error

enumerator kStatus_I3C_S0S1Error

S0 or S1 error

enum _i3c_hdr_mode

I3C HDR modes.

Values:

enumerator kI3C_HDRModeNone

enumerator kI3C_HDRModeDDR

enumerator kI3C_HDRModeTSP

enumerator kI3C_HDRModeTSL

typedef enum _i3c_hdr_mode i3c_hdr_mode_t

I3C HDR modes.

typedef struct _i3c_device_info i3c_device_info_t

I3C device information.

I3C_RETRY_TIMES

Max loops to wait for I3C operation status complete.

This is the maximum number of loops to wait for I3C operation status complete. If set to 0, it will wait indefinitely.

I3C_MAX_DEVCNT

I3C_IBI_BUFF_SIZE

struct _i3c_device_info

#include <fsl_i3c.h> I3C device information.

Public Members

uint8_t dynamicAddr

Device dynamic address.

uint8_t staticAddr

Static address.

uint8_t dcr

Device characteristics register information.

uint8_t bcr

Bus characteristics register information.

uint16_t vendorID
Device vendor ID(manufacture ID).

uint32_t partNumber
Device part number info

uint16_t maxReadLength
Maximum read length.

uint16_t maxWriteLength
Maximum write length.

uint8_t hdrMode
Support hdr mode, could be OR logic in i3c_hdr_mode.

2.31 I3C Common Driver

typedef struct *i3c_config* i3c_config_t

Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

uint32_t I3C_GetInstance(I3C_Type *base)

Get which instance current I3C is used.

Parameters

- base – The I3C peripheral base address.

void I3C_GetDefaultConfig(*i3c_config_t* *config)

Provides a default configuration for the I3C peripheral, the configuration covers both master functionality and slave functionality.

This function provides the following default configuration for I3C:

```
config->enableMaster      = kI3C_MasterCapable;
config->disableTimeout    = false;
config->hKeep             = kI3C_MasterHighKeeperNone;
config->enableOpenDrainStop = true;
config->enableOpenDrainHigh = true;
config->baudRate_Hz.i2cBaud = 400000U;
config->baudRate_Hz.i3cPushPullBaud = 12500000U;
config->baudRate_Hz.i3cOpenDrainBaud = 2500000U;
config->masterDynamicAddress = 0x0AU;
config->slowClock_Hz       = 1000000U;
config->enableSlave        = true;
config->vendorID           = 0x11BU;
config->enableRandomPart   = false;
config->partNumber         = 0;
config->dcr                = 0;
config->bcr = 0;
config->hdrMode            = (uint8_t)kI3C_HDRModeDDR;
config->nakAllRequest       = false;
config->ignoreS0S1Error    = false;
config->offline            = false;
config->matchSlaveStartStop = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the common I3C driver with `I3C_Init()`.

Parameters

- `config` – **[out]** User provided configuration structure for default values. Refer to `i3c_config_t`.

`void I3C_Init(I3C_Type *base, const i3c_config_t *config, uint32_t sourceClock_Hz)`

Initializes the I3C peripheral. This function enables the peripheral clock and initializes the I3C peripheral as described by the user provided configuration. This will initialize both the master peripheral and slave peripheral so that I3C module could work as pure master, pure slave or secondary master, etc. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.
- `config` – User provided peripheral configuration. Use `I3C_GetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`struct _i3c_config`

`#include <fsl_i3c.h>` Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

`i3c_master_enable_t enableMaster`

Enable master mode.

`bool disableTimeout`

Whether to disable timeout to prevent the ERRWARN.

`i3c_master_hkeep_t hKeep`

High keeper mode setting.

`bool enableOpenDrainStop`

Whether to emit open-drain speed STOP.

`bool enableOpenDrainHigh`

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

`i3c_baudrate_hz_t baudRate_Hz`

Desired baud rate settings.

`i3c_start_scl_delay_t startSclDelay`

I3C SCL delay after START.

`i3c_start_scl_delay_t restartSclDelay`

I3C SCL delay after Repeated START.

`uint8_t masterDynamicAddress`

Main master dynamic address configuration.

`uint32_t maxWriteLength`
Maximum write length.

`uint32_t maxReadLength`
Maximum read length.

`bool enableSlave`
Whether to enable slave.

`uint8_t staticAddr`
Static address.

`uint16_t vendorID`
Device vendor ID(manufacture ID).

`uint32_t partNumber`
Device part number info

`uint8_t dcr`
Device characteristics register information.

`uint8_t bcr`
Bus characteristics register information.

`uint8_t hdrMode`
Support hdr mode, could be OR logic in enumeration:`i3c_hdr_mode_t`.

`bool nakAllRequest`
Whether to reply NAK to all requests except broadcast CCC.

`bool ignoreS0S1Error`
Whether to ignore S0/S1 error in SDR mode.

`bool offline`
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

`bool matchSlaveStartStop`
Whether to assert start/stop status only the time slave is addressed.

2.32 I3C Master Driver

`void I3C_MasterGetDefaultConfig(i3c_master_config_t *masterConfig)`

Provides a default configuration for the I3C master peripheral.

This function provides the following default configuration for the I3C master peripheral:

```
masterConfig->enableMaster      = kI3C_MasterOn;
masterConfig->disableTimeout    = false;
masterConfig->hKeep              = kI3C_MasterHighKeeperNone;
masterConfig->enableOpenDrainStop = true;
masterConfig->enableOpenDrainHigh = true;
masterConfig->baudRate_Hz        = 100000U;
masterConfig->busType            = kI3C_TypeI2C;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I3C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_master_config_t`.

```
void I3C_MasterInit(I3C_Type *base, const i3c_master_config_t *masterConfig, uint32_t sourceClock_Hz)
```

Initializes the I3C master peripheral.

This function enables the peripheral clock and initializes the I3C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I3C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

```
void I3C_MasterDeinit(I3C_Type *base)
```

Deinitializes the I3C master peripheral.

This function disables the I3C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterCheckAndClearError(I3C_Type *base, uint32_t status)
```

```
status_t I3C_MasterWaitForCtrlDone(I3C_Type *base, bool waitIdle)
```

```
status_t I3C_CheckForBusyBus(I3C_Type *base)
```

```
static inline void I3C_MasterEnable(I3C_Type *base, i3c_master_enable_t enable)
```

Set I3C module master mode.

Parameters

- `base` – The I3C peripheral base address.
- `enable` – Enable master mode.

```
void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)
```

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- base – The I3C peripheral base address.
- slaveConfig – User provided peripheral configuration. Use I3C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
- slowClock_Hz – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH defines as 1, this parameter is useless.

void I3C_SlaveDeinit(I3C_Type *base)

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- base – The I3C peripheral base address.

static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)

Enable/Disable Slave.

Parameters

- base – The I3C peripheral base address.
- isEnabled – Enable or disable.

static inline uint32_t I3C_MasterGetStatusFlags(I3C_Type *base)

Gets the I3C master status flags.

A bit mask with the state of all I3C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

[_i3c_master_flags](#)

Parameters

- base – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_MasterClearStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C master status flag state.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

Attempts to clear other flags has no effect.

See also:

`_i3c_master_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

```
static inline uint32_t I3C_MasterGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C master error status flags.

A bit mask with the state of all I3C master error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_master_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_MasterClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C master error status flag state.

See also:

`_i3c_master_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_master_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

```
i3c_master_state_t I3C_MasterGetState(I3C_Type *base)
```

Gets the I3C master state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C master state.

```
static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)
```

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

```
static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.

- 0: related status flag is not set.

static inline void I3C__SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

i3c_slave_activity_state_t I3C__SlaveGetActivityState(I3C_Type *base)

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

status_t I3C__SlaveCheckAndClearError(I3C_Type *base, uint32_t status)

static inline void I3C__MasterEnableInterrupts(I3C_Type *base, uint32_t interruptMask)

Enables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline void I3C__MasterDisableInterrupts(I3C_Type *base, uint32_t interruptMask)

Disables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I3C__MasterGetEnabledInterrupts(I3C_Type *base)

Returns the set of currently enabled I3C master interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_MasterGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C master interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

static inline void I3C_MasterEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)

Enables or disables I3C master DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

static inline uint32_t I3C_MasterGetTxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C master transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Transmit Data Register address.

static inline uint32_t I3C_MasterGetRxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C master receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Receive Data Register address.


```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)
```

Enables or disables I3C slave DMA requests.

Parameters

- *base* – The I3C peripheral base address.
- *enableTx* – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- *enableRx* – Enable flag for receive DMA request. Pass true for enable, false for disable.
- *width* – DMA read/write unit in bytes.

```
static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

```
static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave receive data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

```
static inline void I3C_MasterSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl, i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)
```

Sets the watermarks for I3C master FIFOs.

Parameters

- *base* – The I3C peripheral base address.
- *txLvl* – Transmit FIFO watermark level. The `kI3C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- *rxLvl* – Receive FIFO watermark level. The `kI3C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- *flushTx* – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- *flushRx* – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_MasterGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C master FIFOs.

Parameters

- *base* – The I3C peripheral base address.
- *txCount* – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.

- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
static inline void I3C_SlaveSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl,  
                                         i3c_rx_trigger_level_t rxLvl, bool flushTx, bool  
                                         flushRx)
```

Sets the watermarks for I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.
- `rxLvl` – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
void I3C_MasterSetBaudRate(I3C_Type *base, const i3c_baudrate_hz_t *baudRate_Hz, uint32_t  
                           sourceClock_Hz)
```

Sets the I3C bus frequency for master transactions.

The I3C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

- `base` – The I3C peripheral base address.
- `baudRate_Hz` – Pointer to structure of requested bus frequency in Hertz.
- `sourceClock_Hz` – I3C functional clock frequency in Hertz.

```
static inline bool I3C_MasterGetBusIdleState(I3C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

status_t I3C_MasterStartWithRxSize(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir, uint8_t rxSize)

Sends a START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- *base* – The I3C peripheral base address.
- *type* – The bus type to use in this transaction.
- *address* – 7-bit slave device address, in bits [6:0].
- *dir* – Master transfer direction, either *kI3C_Read* or *kI3C_Write*. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- *rxSize* – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

- *kStatus_Success* – START signal and address were successfully enqueued in the transmit FIFO.
- *kStatus_I3C_Busy* – Another master is currently utilizing the bus.

status_t I3C_MasterStart(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir)

Sends a START signal and slave address on the I2C/I3C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- *base* – The I3C peripheral base address.
- *type* – The bus type to use in this transaction.
- *address* – 7-bit slave device address, in bits [6:0].
- *dir* – Master transfer direction, either *kI3C_Read* or *kI3C_Write*. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- *kStatus_Success* – START signal and address were successfully enqueued in the transmit FIFO.
- *kStatus_I3C_Busy* – Another master is currently utilizing the bus.

status_t I3C_MasterRepeatedStartWithRxSize(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir, uint8_t rxSize)

Sends a repeated START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to send a Repeated START signal when a transfer is already in progress. Like *I3C_MasterStart()*, it also sends the specified 7-bit address. Call this API also configures the read terminate size for the following read transfer. For example, set the *rxSize* = 2, the

following read transfer will be terminated after two bytes of data received. Write transfer will not be affected by the rxSize configuration.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- `rxSize` – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
static inline status_t I3C_MasterRepeatedStart(I3C_Type *base, i3c_bus_type_t type, uint8_t  
                                              address, i3c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C/I3C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `I3C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
status_t I3C_MasterSend(I3C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)
```

Performs a polling send transfer on the I2C/I3C bus.

Sends up to `txSize` number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_I3C_Nak`.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Nak` – The slave device sent a NAK in response to an address.
- `kStatus_I3C_WriteAbort` – The slave device sent a NAK in response to a write.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_WriteFifoError` – Write to M/SWDATAB register when FIFO full.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

status_t I3C_MasterReceive(I3C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transfer on the I2C/I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.
- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Term` – The master terminates slave read.
- `kStatus_I3C_HdrParityError` – Parity error from DDR read.
- `kStatus_I3C_CrcError` – CRC error from DDR read.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_ReadFifoError` – Read from M/SRDATAB register when FIFO empty.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

status_t I3C_MasterStop(I3C_Type *base)

Sends a STOP signal on the I2C/I3C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.

- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`void I3C_MasterEmitRequest(I3C_Type *base, i3c_bus_request_t masterReq)`
I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `masterReq` – I3C master request of type `i3c_bus_request_t`

`static inline void I3C_MasterEmitIBIResponse(I3C_Type *base, i3c_ibi_response_t ibiResponse)`
I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `ibiResponse` – I3C master emit IBI response of type `i3c_ibi_response_t`

`void I3C_MasterRegisterIBI(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
I3C master register IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to ibi rule description of type `i3c_register_ibi_addr_t`

`void I3C_MasterGetIBIRules(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
I3C master get IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to store the read out ibi rule description.

`i3c_ibi_type_t I3C_GetIBIType(I3C_Type *base)`
I3C master get IBI Type.

Parameters

- `base` – The I3C peripheral base address.

Return values

`i3c_ibi_type_t` – Type of `i3c_ibi_type_t`.

`static inline uint8_t I3C_GetIBIAddress(I3C_Type *base)`
I3C master get IBI Address.

Parameters

- `base` – The I3C peripheral base address.

Return values

The – 8-bit IBI address.

`status_t I3C_MasterProcessDAASpecifiedBaudrate(I3C_Type *base, uint8_t *addressList, uint32_t count, i3c_master_daa_baudrate_t *daaBaudRate)`

Performs a DAA in the i3c bus with specified temporary baud rate.

Parameters

- `base` – The I3C peripheral base address.

- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list.
- `daaBaudRate` – The temporary baud rate in DAA process, NULL for using initial setting. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus__Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
static inline status_t I3C_MasterProcessDAA(I3C_Type *base, uint8_t *addressList, uint32_t count)
```

Performs a DAA in the i3c bus.

Parameters

- `base` – The I3C peripheral base address.
- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus__Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
i3c_device_info_t *I3C_MasterGetDeviceListAfterDAA(I3C_Type *base, uint8_t *count)
```

Get device information list after DAA process is done.

Parameters

- `base` – The I3C peripheral base address.
- `count` – **[out]** The pointer to store the available device count.

Returns

Pointer to the `i3c_device_info_t` array.

```
void I3C_MasterClearDeviceCount(I3C_Type *base)
```

Clear the global device count which represents current devices number on the bus. When user resets all dynamic addresses on the bus, should call this API.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterTransferBlocking(I3C_Type *base, i3c_master_transfer_t *transfer)
```

Performs a master polling transfer on the I2C/I3C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- `base` – The I3C peripheral base address.
- `transfer` – Pointer to the transfer structure.

Return values

- `kStatus_I3C_Success` – Data was received successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_IBIWon` – The I3C slave event IBI or MR or HJ won the arbitration on a header address.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Nak` – The slave device sent a NAK in response to an address.
- `kStatus_I3C_WriteAbort` – The slave device sent a NAK in response to a write.
- `kStatus_I3C_Term` – The master terminates slave read.
- `kStatus_I3C_HdrParityError` – Parity error from DDR read.
- `kStatus_I3C_CrcError` – CRC error from DDR read.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_ReadFifoError` – Read from M/SRDATA register when FIFO empty.
- `kStatus_I3C_WriteFifoError` – Write to M/SWDATA register when FIFO full.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

void I3C_MasterTransferCreateHandle(I3C_Type *base, *i3c_master_handle_t* *handle, const *i3c_master_transfer_callback_t* *callback, void *userData)

Creates a new handle for the I3C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The I3C peripheral base address.
- handle – **[out]** Pointer to the I3C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I3C_MasterTransferNonBlocking(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C/I3C bus.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_I3C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t I3C_MasterTransferGetCount(I3C_Type *base, *i3c_master_handle_t* *handle, *size_t* *count)

Returns number of bytes transferred so far.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void I3C_MasterTransferAbort(I3C_Type *base, *i3c_master_handle_t* *handle)

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I3C peripheral's IRQ priority.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.

void I3C_MasterTransferHandleIRQ(I3C_Type *base, void *intHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to the I3C master driver handle.

enum _i3c_master_flags

I3C master peripheral flags.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterBetweenFlag

Between messages/DAA's flag

enumerator kI3C_MasterNackDetectFlag

NACK detected flag

enumerator kI3C_MasterSlaveStartFlag

Slave request start flag

enumerator kI3C_MasterControlDoneFlag

Master request complete flag

enumerator kI3C_MasterCompleteFlag

Transfer complete flag

enumerator kI3C_MasterRxReadyFlag

Rx data ready in Rx buffer flag

enumerator kI3C_MasterTxReadyFlag

Tx buffer ready for Tx data flag

enumerator kI3C_MasterArbitrationWonFlag

Header address won arbitration flag

enumerator kI3C_MasterErrorFlag

Error occurred flag

enumerator kI3C_MasterSlave2MasterFlag

Switch from slave to master flag

enumerator kI3C_MasterClearFlags

enum _i3c_master_error_flags

I3C master error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterErrorNackFlag

Slave NACKed the last address

enumerator kI3C_MasterErrorWriteAbortFlag

Slave NACKed the write data

enumerator kI3C_MasterErrorParityFlag

Parity error from DDR read

enumerator kI3C_MasterErrorCrcFlag

CRC error from DDR read

enumerator kI3C_MasterErrorReadFlag

Read from MRDATAB register when FIFO empty

enumerator kI3C_MasterErrorWriteFlag

Write to MWDATAB register when FIFO full

enumerator kI3C_MasterErrorMsgFlag

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kI3C_MasterErrorInvalidReqFlag

Invalid use of request

enumerator kI3C_MasterErrorTimeoutFlag

The module has stalled too long in a frame

enumerator kI3C_MasterAllErrorFlags

All error flags

enum _i3c_master_state

I3C working master state.

Values:

enumerator kI3C_MasterStateIdle

Bus stopped.

enumerator kI3C_MasterStateSlvReq

Bus stopped but slave holding SDA low.

enumerator kI3C_MasterStateMsgSdr

In SDR Message mode from using MWMSG_SDR.

enumerator kI3C_MasterStateNormAct

In normal active SDR mode.

enumerator kI3C_MasterStateDdr

In DDR Message mode.

enumerator kI3C_MasterStateDaa

In ENTDAAs mode.

enumerator kI3C_MasterStateIbiAck

Waiting on IBI ACK/NACK decision.

enumerator kI3C_MasterStateIbiRcv

Receiving IBI.

enum _i3c_master_enable

I3C master enable configuration.

Values:

enumerator kI3C_MasterOff

Master off.

enumerator kI3C_MasterOn

Master on.

enumerator kI3C_MasterCapable

Master capable.

enum _i3c_master_hkeep

I3C high keeper configuration.

Values:

enumerator kI3C_MasterHighKeeperNone

Use PUR to hold SCL high.

enumerator kI3C_MasterHighKeeperWiredIn

Use pin_HK controls.

enumerator kI3C_MasterPassiveSDA

Hi-Z for Bus Free and hold SDA.

enumerator kI3C_MasterPassiveSDASCL

Hi-Z both for Bus Free, and can Hi-Z SDA for hold.

enum _i3c_bus_request

Emits the requested operation when doing in pieces vs. by message.

Values:

enumerator kI3C_RequestNone

No request.

enumerator kI3C_RequestEmitStartAddr

Request to emit start and address on bus.

enumerator kI3C_RequestEmitStop

Request to emit stop on bus.

enumerator kI3C_RequestIbiAckNack

Manual IBI ACK or NACK.

enumerator kI3C_RequestProcessDAA

Process DAA.

enumerator kI3C_RequestForceExit

Request to force exit.

enumerator kI3C_RequestAutoIbi
Hold in stopped state, but Auto-emit START,7E.

enum _i3c_bus_type

Bus type with EmitStartAddr.

Values:

enumerator kI3C_TypeI3CSdr
SDR mode of I3C.

enumerator kI3C_TypeI2C
Standard i2c protocol.

enumerator kI3C_TypeI3CDdr
HDR-DDR mode of I3C.

enum _i3c_ibi_response

IBI response.

Values:

enumerator kI3C_IbiRespAck
ACK with no mandatory byte.

enumerator kI3C_IbiRespNack
NACK.

enumerator kI3C_IbiRespAckMandatory
ACK with mandatory byte.

enumerator kI3C_IbiRespManual
Reserved.

enum _i3c_ibi_type

IBI type.

Values:

enumerator kI3C_IbiNormal
In-band interrupt.

enumerator kI3C_IbiHotJoin
slave hot join.

enumerator kI3C_IbiMasterRequest
slave master ship request.

enum _i3c_ibi_state

IBI state.

Values:

enumerator kI3C_IbiReady
In-band interrupt ready state, ready for user to handle.

enumerator kI3C_IbiDataBuffNeed
In-band interrupt need data buffer for data receive.

enumerator kI3C_IbiAckNackPending
In-band interrupt Ack/Nack pending for decision.

enum _i3c_direction

Direction of master and slave transfers.

Values:

enumerator kI3C_Write

Master transmit.

enumerator kI3C_Read

Master receive.

enum _i3c_tx_trigger_level

Watermark of TX int/dma trigger level.

Values:

enumerator kI3C_TxTriggerOnEmpty

Trigger on empty.

enumerator kI3C_TxTriggerUntilOneQuarterOrLess

Trigger on 1/4 full or less.

enumerator kI3C_TxTriggerUntilOneHalfOrLess

Trigger on 1/2 full or less.

enumerator kI3C_TxTriggerUntilOneLessThanFull

Trigger on 1 less than full or less.

enum _i3c_rx_trigger_level

Watermark of RX int/dma trigger level.

Values:

enumerator kI3C_RxTriggerOnNotEmpty

Trigger on not empty.

enumerator kI3C_RxTriggerUntilOneQuarterOrMore

Trigger on 1/4 full or more.

enumerator kI3C_RxTriggerUntilOneHalfOrMore

Trigger on 1/2 full or more.

enumerator kI3C_RxTriggerUntilThreeQuarterOrMore

Trigger on 3/4 full or more.

enum _i3c_rx_term_ops

I3C master read termination operations.

Values:

enumerator kI3C_RxTermDisable

Master doesn't terminate read, used for CCC transfer.

enumerator kI3C_RxAutoTerm

Master auto terminate read after receiving specified bytes(<=255).

enumerator kI3C_RxTermLastByte

Master terminates read at any time after START, no length limitation.

enum _i3c_start_scl_delay

I3C start SCL delay options.

Values:

enumerator kI3C_NoDelay

No delay.

enumerator kI3C_IncreaseSclHalfPeriod

Increases SCL clock period by 1/2.

enumerator kI3C_IncreaseSclOnePeriod

Increases SCL clock period by 1.

enumerator kI3C_IncreaseSclOneAndHalfPeriod

Increases SCL clock period by 1 1/2

enum _i3c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_i3c_master_transfer::flags` field.

Values:

enumerator kI3C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kI3C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kI3C_TransferRepeatedStartFlag

Send a repeated start condition

enumerator kI3C_TransferNoStopFlag

Don't send a stop condition.

enumerator kI3C_TransferWordsFlag

Transfer in words, else transfer in bytes.

enumerator kI3C_TransferDisableRxTermFlag

Disable Rx termination. Note: It's for I3C CCC transfer.

enumerator kI3C_TransferRxAutoTermFlag

Set Rx auto-termination. Note: It's adaptive based on Rx size(<=255 bytes) except in I3C_MasterReceive.

enumerator kI3C_TransferStartWithBroadcastAddr

Start transfer with 0x7E, then read/write data with device address.

typedef enum _i3c_master_state i3c_master_state_t

I3C working master state.

typedef enum _i3c_master_enable i3c_master_enable_t

I3C master enable configuration.

typedef enum _i3c_master_hkeep i3c_master_hkeep_t

I3C high keeper configuration.

typedef enum _i3c_bus_request i3c_bus_request_t

Emits the requested operation when doing in pieces vs. by message.

typedef enum _i3c_bus_type i3c_bus_type_t

Bus type with EmitStartAddr.

```
typedef enum _i3c_ibi_response i3c_ibi_response_t
    IBI response.
```

```
typedef enum _i3c_ibi_type i3c_ibi_type_t
    IBI type.
```

```
typedef enum _i3c_ibi_state i3c_ibi_state_t
    IBI state.
```

```
typedef enum _i3c_direction i3c_direction_t
    Direction of master and slave transfers.
```

```
typedef enum _i3c_tx_trigger_level i3c_tx_trigger_level_t
    Watermark of TX int/dma trigger level.
```

```
typedef enum _i3c_rx_trigger_level i3c_rx_trigger_level_t
    Watermark of RX int/dma trigger level.
```

```
typedef enum _i3c_rx_term_ops i3c_rx_term_ops_t
    I3C master read termination operations.
```

```
typedef enum _i3c_start_scl_delay i3c_start_scl_delay_t
    I3C start SCL delay options.
```

```
typedef struct _i3c_register_ibi_addr i3c_register_ibi_addr_t
    Structure with setting master IBI rules and slave registry.
```

```
typedef struct _i3c_baudrate i3c_baudrate_hz_t
    Structure with I3C baudrate settings.
```

```
typedef struct _i3c_master_daa_baudrate i3c_master_daa_baudrate_t
    I3C DAA baud rate configuration.
```

```
typedef struct _i3c_master_config i3c_master_config_t
    Structure with settings to initialize the I3C master module.
```

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_MasterGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef struct _i3c_master_transfer i3c_master_transfer_t
```

```
typedef struct _i3c_master_handle i3c_master_handle_t
```

```
typedef struct _i3c_master_transfer_callback i3c_master_transfer_callback_t
    i3c master callback functions.
```

```
typedef void (*i3c_master_isr_t)(I3C_Type *base, void *handle)
    Typedef for master interrupt handler.
```

```
struct _i3c_register_ibi_addr
    #include <fsl_i3c.h> Structure with setting master IBI rules and slave registry.
```

Public Members

```
uint8_t address[5]
    Address array for registry.
```

```
bool i3cFastStart
    Allow the START header to run as push-pull speed if all dynamic addresses take MSB 0.
```


bool ibiHasPayload

Whether the address array has mandatory IBI byte.

struct __i3c__baudrate

#include <fsl_i3c.h> Structure with I3C baudrate settings.

Public Members

uint32_t i2cBaud

Desired I2C baud rate in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct __i3c__master__daa__baudrate

#include <fsl_i3c.h> I3C DAA baud rate configuration.

Public Members

uint32_t sourceClock_Hz

FCLK, function clock in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct __i3c__master__config

#include <fsl_i3c.h> Structure with settings to initialize the I3C master module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i3c_master_enable_t enableMaster

Enable master mode.

bool disableTimeout

Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep

High keeper mode setting.

bool enableOpenDrainStop

Whether to emit open-drain speed STOP.

bool enableOpenDrainHigh

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate_Hz

Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay

I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay

I3C SCL delay after Repeated START.

struct *_i3c_master_transfer_callback*

#include <fsl_i3c.h> i3c master callback functions.

Public Members

void (*slave2Master)(I3C_Type *base, void *userData)

Transfer complete callback

void (*ibiCallback)(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_ibi_type_t* ibiType, *i3c_ibi_state_t* ibiState)

IBI event callback

void (*transferComplete)(I3C_Type *base, *i3c_master_handle_t* *handle, *status_t* completionStatus, void *userData)

Transfer complete callback

struct *_i3c_master_transfer*

#include <fsl_i3c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I3C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration *_i3c_master_transfer_flags* for available options. Set to 0 or *kI3C_TransferDefaultFlag* for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i3c_direction_t direction

Either *kI3C_Read* or *kI3C_Write*.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

i3c_bus_type_t busType

bus type.

i3c_ibi_response_t ibiResponse

ibi response during transfer.

```
struct _i3c_master_handle
    #include <fsl_i3c.h> Driver handle for master non-blocking APIs.
```

Note: The contents of this structure are private and subject to change.

Public Members

```
uint8_t state
    Transfer state machine current state.

uint32_t remainingBytes
    Remaining byte count in current state.

i3c_rx_term_ops_t rxTermOps
    Read termination operation.

i3c_master_transfer_t transfer
    Copy of the current transfer info.

uint8_t ibiAddress
    Slave address which request IBI.

uint8_t *ibiBuff
    Pointer to IBI buffer to keep ibi bytes.

size_t ibiPayloadSize
    IBI payload size.

i3c_ibi_type_t ibiType
    IBI type.

i3c_master_transfer_callback_t callback
    Callback functions pointer.

void *userData
    Application data passed to callback.
```

2.33 I3C Slave Driver

```
void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)
    Provides a default configuration for the I3C slave peripheral.
```

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t
    slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I3C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I3C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `slowClock_Hz` – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If `FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH` defines as 1, this parameter is useless.

```
void I3C_SlaveDeinit(I3C_Type *base)
```

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- `base` – The I3C peripheral base address.

```
static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)
```

Enable/Disable Slave.

Parameters

- `base` – The I3C peripheral base address.
- `isEnabled` – Enable or disable.

```
static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)
```

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

`static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)`

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

`static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)`

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

`i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)`

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

`status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)`

```
static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C slave interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)
```

Enables or disables I3C slave DMA requests.

Parameters

- `base` – The I3C peripheral base address.
- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.
- `width` – DMA read/write unit in bytes.

```
static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

```
static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave receive data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

```
static inline void I3C_SlaveSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl, i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)
```

Sets the watermarks for I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.

- `rxLvl` – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

`static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)`

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_SlaveTransferCreateHandle(I3C_Type *base, i3c_slave_handle_t *handle, i3c_slave_transfer_callback_t callback, void *userData)`

Creates a new handle for the I3C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_SlaveTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C slave driver handle.

- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_SlaveTransferNonBlocking(I3C_Type *base, i3c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling `I2C_SlaveInit()` and `I3C_SlaveTransferCreateHandle()` to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to `I3C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i3c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI3C_SlaveTransmitEvent` and `kI3C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI3C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i3c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI3C_SlaveAllEvents` to enable all events.

Return values

- `kStatus__Success` – Slave transfers were successfully started.
- `kStatus_I3C_Busy` – Slave transfers have already been started on this handle.

`status_t I3C_SlaveTransferGetCount(I3C_Type *base, i3c_slave_handle_t *handle, size_t *count)`

Gets the slave transfer status during a non-blocking transfer.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure.
- `count` – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- `kStatus__Success` –
- `kStatus_NoTransferInProgress` –

`void I3C_SlaveTransferAbort(I3C_Type *base, i3c_slave_handle_t *handle)`

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- `base` – The I3C peripheral base address.

- handle – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`void I3C_SlaveTransferHandleIRQ(I3C_Type *base, void *intHandle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`enum _i3c_slave_flags`

I3C slave peripheral flags.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator `kI3C_SlaveNotStopFlag`

Slave status not stop flag

enumerator `kI3C_SlaveMessageFlag`

Slave status message, indicating slave is listening to the bus traffic or responding

enumerator `kI3C_SlaveRequiredReadFlag`

Slave status required, either is master doing SDR read from slave, or is IBI pushing out.

enumerator kI3C_SlaveRequiredWriteFlag
Slave status request write, master is doing SDR write to slave, except slave in ENTDAAMode

enumerator kI3C_SlaveBusDAAFlag
I3C bus is in ENTDAAMode

enumerator kI3C_SlaveBusHDRModeFlag
I3C bus is in HDR mode

enumerator kI3C_SlaveBusStartFlag
Start/Re-start event is seen since the bus was last cleared

enumerator kI3C_SlaveMatchedFlag
Slave address(dynamic/static) matched since last cleared

enumerator kI3C_SlaveBusStopFlag
Stop event is seen since the bus was last cleared

enumerator kI3C_SlaveRxReadyFlag
Rx data ready in rx buffer flag

enumerator kI3C_SlaveTxReadyFlag
Tx buffer ready for Tx data flag

enumerator kI3C_SlaveDynamicAddrChangedFlag
Slave dynamic address has been assigned, re-assigned, or lost

enumerator kI3C_SlaveReceivedCCCFlag
Slave received Common command code

enumerator kI3C_SlaveErrorFlag
Error occurred flag

enumerator kI3C_SlaveHDRCommandMatchFlag
High data rate command match

enumerator kI3C_SlaveCCCHandledFlag
Slave received Common command code is handled by I3C module

enumerator kI3C_SlaveEventSentFlag
Slave IBI/P2P/MR/HJ event has been sent

enumerator kI3C_SlaveIbiDisableFlag
Slave in band interrupt is disabled.

enumerator kI3C_SlaveMasterRequestDisabledFlag
Slave master request is disabled.

enumerator kI3C_SlaveHotJoinDisabledFlag
Slave Hot-Join is disabled.

enumerator kI3C_SlaveClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kI3C_SlaveAllIrqFlags

enum _i3c_slave_error_flags
I3C slave error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_SlaveErrorOverrunFlag
Slave internal from-bus buffer/FIFO overrun.

enumerator kI3C_SlaveErrorUnderrunFlag
Slave internal to-bus buffer/FIFO underrun

enumerator kI3C_SlaveErrorUnderrunNakFlag
Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kI3C_SlaveErrorTermFlag
Terminate error from master

enumerator kI3C_SlaveErrorInvalidStartFlag
Slave invalid start flag

enumerator kI3C_SlaveErrorSdrParityFlag
SDR parity error

enumerator kI3C_SlaveErrorHdrParityFlag
HDR parity error

enumerator kI3C_SlaveErrorHdrCRCFlag
HDR-DDR CRC error

enumerator kI3C_SlaveErrorS0S1Flag
S0 or S1 error

enumerator kI3C_SlaveErrorOverreadFlag
Over-read error

enumerator kI3C_SlaveErrorOverwriteFlag
Over-write error

enum _i3c_slave_event
I3C slave.event.

Values:

enumerator kI3C_SlaveEventNormal
Normal mode.

enumerator kI3C_SlaveEventIBI
In band interrupt event.

enumerator kI3C_SlaveEventMasterReq
Master request event.

enumerator kI3C_SlaveEventHotJoinReq
Hot-join event.

enum _i3c_slave_activity_state
I3C slave.activity state.

Values:

enumerator kI3C_SlaveNoLatency
Normal bus operation

enumerator kI3C_SlaveLatency1Ms
1ms of latency.

enumerator kI3C_SlaveLatency100Ms
100ms of latency.

enumerator kI3C_SlaveLatency10S
10s latency.

enum _i3c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kI3C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kI3C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kI3C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveRequiredTransmitEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveStartEvent

A start/repeated start was detected.

enumerator kI3C_SlaveHDRCommandMatchEvent

Slave Match HDR Command.

enumerator kI3C_SlaveCompletionEvent

A stop was detected, completing the transfer.

enumerator kI3C_SlaveRequestSentEvent

Slave request event sent.

enumerator kI3C_SlaveReceivedCCCEvent

Slave received CCC event, need to handle by application.

enumerator kI3C_SlaveAllEvents

Bit mask of all available events.

typedef enum _i3c_slave_event i3c_slave_event_t
I3C slave.event.

typedef enum _i3c_slave_activity_state i3c_slave_activity_state_t
I3C slave.activity state.

typedef struct _i3c_slave_config i3c_slave_config_t

Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _i3c_slave_transfer_event i3c_slave_transfer_event_t
```

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

```
typedef struct _i3c_slave_transfer i3c_slave_transfer_t
```

I3C slave transfer structure.

```
typedef struct _i3c_slave_handle i3c_slave_handle_t
```

```
typedef void (*i3c_slave_transfer_callback_t)(I3C_Type *base, i3c_slave_transfer_t *transfer, void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the I3C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the I3C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef void (*i3c_slave_isr_t)(I3C_Type *base, void *handle)
```

Typedef for slave interrupt handler.

```
struct _i3c_slave_config
```

#include <fsl_i3c.h> Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableSlave

Whether to enable slave.

uint8_t staticAddr

Static address.

uint16_t vendorID

Device vendor ID(manufacture ID).

uint32_t partNumber

Device part number info

uint8_t dcr

Device characteristics register information.

`uint8_t bcr`
Bus characteristics register information.

`uint8_t hdrMode`
Support hdr mode, could be OR logic in enumeration:`i3c_hdr_mode_t`.

`bool nakAllRequest`
Whether to reply NAK to all requests except broadcast CCC.

`bool ignoreS0S1Error`
Whether to ignore S0/S1 error in SDR mode.

`bool offline`
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

`bool matchSlaveStartStop`
Whether to assert start/stop status only the time slave is addressed.

`uint32_t maxWriteLength`
Maximum write length.

`uint32_t maxReadLength`
Maximum read length.

`struct __i3c_slave_transfer`
#include <fsl_i3c.h> I3C slave transfer structure.

Public Members

`uint32_t event`
Reason the callback is being invoked.

`uint8_t *txData`
Transfer buffer

`size_t txDataSize`
Transfer size

`uint8_t *rxData`
Transfer buffer

`size_t rxDataSize`
Transfer size

`status_t completionStatus`
Success or error code describing how the transfer completed. Only applies for `kI3C_SlaveCompletionEvent`.

`size_t transferredCount`
Number of bytes actually transferred since start or last repeated start.

`struct __i3c_slave_handle`
#include <fsl_i3c.h> I3C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

i3c_slave_transfer_t transfer
I3C slave transfer copy.

bool isBusy
Whether transfer is busy.

bool wasTransmit
Whether the last transfer was a transmit.

uint32_t eventMask
Mask of enabled events.

uint32_t transferredCount
Count of bytes transferred.

i3c_slave_transfer_callback_t callback
Callback function called at transfer event.

void *userData
Callback parameter passed to callback.

size_t txFifoSize
Tx Fifo size

2.34 IMU: Inter CPU Messaging Unit

status_t IMU_Init(imu_link_t link)

Initializes the IMU module.

This function sets IMU to initialized state, including:

- Flush the send FIFO.
- Unlock the send FIFO.
- Set the water mark to (IMU_MAX_MSG_FIFO_WATER_MARK)
- Flush the receive FIFO.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout if flushing the receive FIFO takes too long.

Parameters

- link – IMU link.

Return values

- kStatus_Success – The IMU was initialized successfully.
- kStatus_InvalidArgument – Invalid link parameter.
- kStatus_Timeout – Timeout occurred while flushing the receive FIFO.

Returns

status_t


```
void IMU__Deinit(imu_link_t link)
```

De-initializes the IMU module.

Parameters

- link – IMU link.

```
static inline void IMU__WriteMsg(imu_link_t link, uint32_t msg)
```

Write one message to TX FIFO.

This function writes message to the TX FIFO, user need to make sure there is empty space in the TX FIFO, and TX FIFO not locked before calling this function.

Parameters

- link – IMU link.
- msg – The message to send.

```
static inline uint32_t IMU__ReadMsg(imu_link_t link)
```

Read one message from RX FIFO.

User need to make sure there is available message in the RX FIFO.

Parameters

- link – IMU link.

Returns

The message.

```
int32_t IMU__SendMsgsBlocking(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool lockSendFifo)
```

Blocking to send messages.

This function blocks until all messages have been filled to TX FIFO.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message to TX FIFO.
- To lock TX FIFO after filling all messages, set lockSendFifo to true.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return IMU_ERR_TIMEOUT if waiting for FIFO space takes too long.

Parameters

- link – IMU link.
- msgs – The messages to send.
- msgCount – Message count, one message is a 32-bit word.
- lockSendFifo – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED. If a timeout occurs while waiting for FIFO space, it returns IMU_ERR_TIMEOUT. Otherwise, this function returns the actual message count sent out, which equals msgCount because this function is blocking until all messages have been filled into TX FIFO or a timeout occurs.

```
int32_t IMU__TrySendMsgs(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool
                        lockSendFifo)
```

Try to send messages.

This function is similar with IMU_SendMsgsBlocking, the difference is, this function tries to send as many as possible, if there is not enough empty slot in TX FIFO, this function fills messages to available empty slots and returns how many messages have been filled.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function fills messages to TX FIFO empty slot, and returns how many messages have been filled.
- If lockSendFifo is set to true, TX FIFO is locked after all messages have been filled to TX FIFO. In other word, TX FIFO is locked if the function return value equals msgCount, when lockSendFifo set to true.

Parameters

- link – IMU link.
- msgs – The messages to send.
- msgCount – Message count, one message is a 32-bit word.
- lockSendFifo – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED, otherwise, this function returns the actual message count sent out.

```
int32_t IMU__TryReceiveMsgs(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount, bool
                           *needAckLock)
```

Try to receive messages.

This function tries to read messages from RX FIFO. It reads the messages already exists in RX FIFO and returns the actual read count.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function the messages in RX FIFO and returns the actual read count.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets needAckLock to true to inform upper layer.

Parameters

- link – IMU link.
- msgs – The buffer to read messages.
- desiredMsgCount – Desired read count, one message is a 32-bit word.
- needAckLock – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns

Count of messages actually received.

```
int32_t IMU_ReceiveMsgsBlocking(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount,  
                               bool *needAckLock)
```

Blocking to receive messages.

This function blocks until all desired messages have been received or the RX FIFO is locked.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function waits for the new messages.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets `needAckLock` to true to inform upper layer.

Parameters

- `link` – IMU link.
- `msgs` – The buffer to read messages.
- `desiredMsgCount` – Desired read count, one message is a 32-bit word.
- `needAckLock` – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns

Count of messages actually received.

```
int32_t IMU_SendMsgPtrBlocking(imu_link_t link, uint32_t msgPtr, bool lockSendFifo)
```

Blocking to send messages pointer.

Compared with `IMU_SendMsgsBlocking`, this function fills message pointer to TX FIFO, but not the message content.

This function blocks until the message pointer is filled to TX FIFO.

- If the TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message pointer to TX FIFO.
- To lock TX FIFO after filling the message pointer, set `lockSendFifo` to true.

If `IMU_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `IMU_ERR_TIMEOUT` if waiting for FIFO space takes too long.

Parameters

- `link` – IMU link.
- `msgPtr` – The buffer pointer to message to send.
- `lockSendFifo` – If set to true, the TX FIFO is locked after message pointer filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`. If a timeout occurs while waiting for FIFO space, it returns `IMU_ERR_TIMEOUT`. Otherwise, this function returns 0 to indicate success.

```
static inline void IMU__LockSendFifo(imu_link_t link, bool lock)
```

Lock or unlock the TX FIFO.

Parameters

- link – IMU link.
- lock – Use true to lock the FIFO, use false to unlock.

```
void IMU__FlushSendFifo(imu_link_t link)
```

Flush the send FIFO.

Flush all messages in send FIFO.

Parameters

- link – IMU link.

```
static inline void IMU__SetSendFifoWaterMark(imu_link_t link, uint8_t waterMark)
```

Set send FIFO water mark.

The water mark must be less than IMU_MAX_MSG_FIFO_WATER_MARK, i.e. $0 < \text{waterMark} \leq \text{IMU\_MAX\_MSG\_FIFO\_WATER\_MARK}$.

Parameters

- link – IMU link.
- waterMark – Send FIFO water mark.

```
static inline uint32_t IMU__GetReceivedMsgCount(imu_link_t link)
```

Get the message count in receive FIFO.

Parameters

- link – IMU link.

Returns

The message count in receive FIFO.

```
static inline uint32_t IMU__GetSendFifoEmptySpace(imu_link_t link)
```

Get the empty slot in send FIFO.

Parameters

- link – IMU link.

Returns

The empty slot count in send FIFO.

```
uint32_t IMU__GetStatusFlags(imu_link_t link)
```

Gets the IMU status flags.

Parameters

- link – IMU link.

Returns

Bit mask of the IMU status flags, see `_imu_status_flags`.

```
static inline void IMU__ClearPendingInterrupts(imu_link_t link, uint32_t mask)
```

Clear the IMU IRQ.

Parameters

- link – IMU link.
- mask – Bit mask of the interrupts to clear, see `_imu_interrupts`.

FSL_IMU_DRIVER_VERSION

IMU driver version.

enum _imu_status_flags

IMU status flags. .

Values:

enumerator kIMU_TxFifoEmpty

enumerator kIMU_TxFifoFull

enumerator kIMU_TxFifoAlmostFull

enumerator kIMU_TxFifoLocked

enumerator kIMU_RxFifoEmpty

enumerator kIMU_RxFifoFull

enumerator kIMU_RxFifoAlmostFull

enumerator kIMU_RxFifoLocked

enum _imu_interrupts

IMU interrupt. .

Values:

enumerator kIMU_RxMsgReadyInterrupt

enumerator kIMU_TxFifoSpaceAvailableInterrupt

IMU_MSG_FIFO_STATUS_MSG_FIFO_LOCKED_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_ALMOST_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_EMPTY_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_SHIFT

IMU_MSG_FIFO_STATUS_WR_PTR_MASK

IMU_MSG_FIFO_STATUS_WR_PTR_SHIFT

IMU_MSG_FIFO_STATUS_RD_PTR_MASK

IMU_MSG_FIFO_STATUS_RD_PTR_SHIFT

IMU_MSG_FIFO_CNTL_MSG_RDY_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_SP_AV_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_FIFO_FLUSH_MASK

IMU_MSG_FIFO_CNTL_WAIT_FOR_ACK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_SHIFT

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK(x)

IMU_WR_MSG(link, msg)

IMU_RD_MSG(link)

IMU_RX_FIFO_LOCKED(link)

IMU_TX_FIFO_LOCKED(link)

IMU_TX_FIFO_ALMOST_FULL(link)

IMU_RX_FIFO_EMPTY(link)

Get Rx FIFO empty status.

IMU_LOCK_TX_FIFO(link)

IMU_UNLOCK_TX_FIFO(link)

IMU_RX_FIFO_MSG_COUNT(link)

IMU_TX_FIFO_MSG_COUNT(link)

IMU_RX_FIFO_MSG_COUNT_FROM_STATUS(rxFifoStatus)

IMU_RX_FIFO_LOCKED_FROM_STATUS(rxFifoStatus)

IMU_TX_FIFO_STATUS(link)

IMU_RX_FIFO_STATUS(link)

IMU_TX_FIFO_CNTL(link)

IMU_ERR_TX_FIFO_LOCKED

IMU driver returned error value.

IMU_ERR_TIMEOUT

IMU driver returned error value timeout.

IMU_MSG_FIFO_MAX_COUNT

Maximum message numbers in FIFO.

IMU_MAX_MSG_FIFO_WATER_MARK

Maximum message FIFO water mark.

IMU_FIFO_SW_WRAPAROUND(ptr)

IMU_WR_PTR(link)

IMU_RD_PTR(link)

IMU_BUSY_POLL_COUNT

Maximum polling iterations for IMU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the IMU driver code before timing out and returning an error.

It applies to all waiting loops in IMU driver.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if sensors don't respond or if communication interfaces fail.

struct IMU_Type

#include <fsl_imu.h> IMU register structure.

2.35 Common Driver

FSL_COMMON_DRIVER_VERSION

common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE

No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART

Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART

Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI

Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(*addr*, *val*)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(*addr*, *val*)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(*addr*, *bits*)

Set the bits 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_CODEC`
Group number for codec status codes.

enumerator `kStatusGroup_ASRC`
Group number for codec status ASRC.

enumerator `kStatusGroup_OTFAD`
Group number for codec status codes.

enumerator `kStatusGroup_SDIOSLV`
Group number for SDIOSLV status codes.

enumerator `kStatusGroup_MECC`
Group number for MECC status codes.

enumerator `kStatusGroup_ENET_QOS`
Group number for ENET_QOS status codes.

enumerator `kStatusGroup_LOG`
Group number for LOG status codes.

enumerator `kStatusGroup_I3CBUS`
Group number for I3CBUS status codes.

enumerator `kStatusGroup_QSCI`
Group number for QSCI status codes.

enumerator `kStatusGroup_ELEMU`
Group number for ELEMU status codes.

enumerator kStatusGroup_QUEUEDSPI
Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
Group number for A-format status codes.

enumerator kStatusGroup_LPC_QSPI
Group number for LPC QSPI status codes.

Generic status return codes.

Values:

enumerator kStatus_Success

Generic status for Success.

enumerator kStatus_Fail

Generic status for Fail.

enumerator kStatus_ReadOnly

Generic status for read only failure.

enumerator kStatus_OutOfRange

Generic status for out of range access.

enumerator kStatus_InvalidArgument

Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

void SDK_Free(void *ptr)

Free memory.

Parameters

- ptr – The memory to be release.

void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- delayTime_us – Delay time in unit of microsecond.
- coreClock_Hz – Core clock frequency with Hz.

static inline *status_t* EnableIRQ(IRQn_Type interrupt)

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt enabled successfully
- kStatus_Fail – Failed to enable the interrupt

static inline *status_t* DisableIRQ(IRQn_Type interrupt)

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt disabled successfully
- kStatus_Fail – Failed to disable the interrupt

static inline *status_t* EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the

LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline status_t IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t newValue)

static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused		Major Version		Minor Version		Bug Fix
31	25 24	17 16	9 8	0		

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there 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.

2.36 LIN: Local Interconnect Network Driver

2.37 LIN Driver

uint32_t LIN_CalcMaxHeaderTimeoutCnt(uint32_t baudRate)

Calculates maximal header time lenght.

Theader_Maximum = 1.4 * THeader_Nominal, THeader_Nominal = 34 * TBit, (13 nominal bits of break; 1 nominal bit of break delimiter; 10 bits for SYNC and 10 bits of PID) The function is not include time for conveying break and break delimiter TIME_OUT_UNIT is in micro second

Parameters

- baudRate – baudrate

uint32_t LIN_CalcMaxResTimeoutCnt(uint32_t baudRate, uint8_t size)

Calculates maximal header time length.

TResponse_Maximum = 1.4 * TResponse_Nominal, TResponse_Nominal = 10 * (NData+ 1) * TBit

Parameters

- baudRate – Baudrate
- size – Frame size

lin_status_t LIN_SetResponse(uint8_t instance, uint8_t *response_buff, uint8_t response_length, uint8_t max_frame_res_timeout)

Forwards a response to a lower level.

Parameters

- instance – LPUART instance
- response_buff – response message
- response_length – length of response
- max_frame_res_timeout – maximal timeout duration for message

Returns

An error code or *lin_status_t*

lin_status_t LIN_RxResponse(uint8_t instance, uint8_t *response_buff, uint8_t response_length, uint8_t max_frame_res_timeout)

Forwards a response to a higher level.

Parameters

- instance – LPUART instance
- response_buff – response message
- response_length – length of response
- max_frame_res_timeout – maximal timeout duration for message

Returns

An error code or *lin_status_t*

lin_status_t LIN_IgnoreResponse(uint8_t instance)

Put a node into idle state.

Parameters

- instance – LPUART instance

Returns

An error code or *lin_status_t*

void LIN_GetSlaveDefaultConfig(*lin_user_config_t* *linUserConfig)

Initializes linUserConfig variable for a slave node.

Parameters

- linUserConfig – Pointer to LIN user config structure

void LIN_GetMasterDefaultConfig(*lin_user_config_t* *linUserConfig)

Initializes linUserConfig variable for a master node.

Parameters

- linUserConfig – Pointer to LIN user config structure

void LIN_CalculateBaudrate(uint32_t instance, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)

Calculates baudrate registers values for given baudrate.

Parameters

- instance – LPUART instance

- baudRate_Bps – LPUART baudrate
- srcClock_Hz – LPUART clock frequency
- osr – LPUART baudrate OSR value, return value
- sbr – LPUART baudrate SBR value, return value

void LIN_SetBaudrate(uint32_t instance, uint32_t osr, uint16_t sbr)

Set baudrate registers values.

Parameters

- instance – LPUART instance
- osr – LPUART baudrate OSR value
- sbr – LPUART baudrate SBR value

lin_status_t LIN_Init(uint32_t instance, *lin_user_config_t* *linUserConfig, *lin_state_t* *linCurrentState, uint32_t clockSource)

Initializes an instance LIN Hardware Interface for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN Hardware Interface clock source in the application to initialize the LIN Hardware Interface.

Parameters

- instance – LPUART instance
- linUserConfig – user configuration structure of type *lin_user_config_t*
- linCurrentState – pointer to the LIN Hardware Interface driver state structure
- clockSource – clock source frequency in Hz for the LIN Hardware Interface

Returns

An error code or *lin_status_t*

lin_status_t LIN_Deinit(uint32_t instance)

Shuts down the LIN Hardware Interface by disabling interrupts and transmitter/receiver.

Parameters

- instance – LPUART instance

Returns

An error code or *lin_status_t*

lin_callback_t LIN_InstallCallback(uint32_t instance, *lin_callback_t* function)

Installs callback function that is used for LIN_DRV_IRQHandler.

Note: After a callback is installed, it bypasses part of the LIN Hardware Interface IRQHandler logic. Therefore, the callback needs to handle the indexes of txBuff and txSize.

Parameters

- instance – LPUART instance.
- function – the LIN receive callback function.

Returns

Former LIN callback function pointer.

lin_status_t LIN_SendFrameDataBlocking(uint32_t instance, const uint8_t *txBuff, uint8_t txSize, uint32_t timeoutMSec)

Sends Frame data out through the LIN Hardware Interface using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- instance – LPUART instance
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or *lin_status_t* LIN_BUS_BUSY if the bus is currently busy, transmission cannot be started.

lin_status_t LIN_SendFrameData(uint32_t instance, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN Hardware Interface using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Note: If users use LIN_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_SetTimeoutCounter(instance, timeoutValue). The timeout value should be big enough to complete the transmission. Timeout in real time is (timeoutValue) * (time period that LIN_TimeoutService is called). For example, if LIN_TimeoutService is called in an timer interrupt with period of 500 micro seconds, then timeout in real time is timeoutValue * 500 micro seconds.

Parameters

- instance – LPUART instance
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or *lin_status_t* LIN_BUS_BUSY if the bus is currently busy, transmission cannot be started. LIN_SUCCESS if the transmission was completed.

lin_status_t LIN_GetTransmitStatus(uint32_t instance, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- instance – LPUART instance
- bytesRemaining – Number of bytes still needed to transmit

Returns

lin_status_t LIN_TX_BUSY if the transmission is still in progress.
LIN_TIMEOUT if timeout occurred and transmission was not completed.
LIN_SUCCESS if the transmission was successful.

lin_status_t LIN_ReceiveFrameDataBlocking(uint32_t instance, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN Hardware Interface using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- instance – LPUART instance
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns

An error code or *lin_status_t*

lin_status_t LIN_ReceiveFrameData(uint32_t instance, uint8_t *rxBuff, uint8_t rxSize)

Receives frame data through the LIN Hardware Interface using non- blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Note: If users use LIN_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_SetTimeoutCounter(instance, timeoutValue). The timeout value should be big enough to complete the reception. Timeout in real time is (timeoutValue) * (time period that LIN_TimeoutService is called). For example, if LIN_TimeoutService is called in an timer interrupt with period of 500 micro seconds, then timeout in real time is timeoutValue * 500 micro seconds.

Parameters

- instance – LPUART instance
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive

Returns

An error code or *lin_status_t*

lin_status_t LIN_AbortTransferData(uint32_t instance)

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- instance – LPUART instance

Returns

An error code or *lin_status_t*

lin_status_t LIN_GetReceiveStatus(uint32_t instance, uint8_t *bytesRemaining)

Get status of an on-going non-blocking reception While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function return the current event ID, LIN_RX_BUSY while receiving and return LIN_SUCCESS, or timeout (LIN_TIMEOUT) when the reception is complete. The bytesRemaining shows number of bytes that still needed to receive.

Parameters

- instance – LPUART instance

- bytesRemaining – Number of bytes still needed to receive

Returns

lin_status_t LIN_RX_BUSY, LIN_TIMEOUT or LIN_SUCCESS

lin_status_t LIN_GoToSleepMode(uint32_t instance)

Puts current LIN node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_MODE.

Parameters

- instance – LPUART instance

Returns

An error code or lin_status_t

lin_status_t LIN_GotoIdleState(uint32_t instance)

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

Parameters

- instance – LPUART instance

Returns

An error code or lin_status_t

lin_status_t LIN_SendWakeupSignal(uint32_t instance)

Sends a wakeup signal through the LIN Hardware Interface.

Parameters

- instance – LPUART instance

Returns

An error code or lin_status_t

lin_node_state_t LIN_GetCurrentNodeState(uint32_t instance)

Get the current LIN node state.

Parameters

- instance – LPUART instance

Returns

current LIN node state

void LIN_TimeoutService(uint32_t instance)

Callback function for Timer Interrupt Handler Users may use (optional, not required) LIN_TimeoutService to check if timeout has occurred during non-blocking frame data transmission and reception. User may initialize a timer (for example FTM) in Output Compare Mode with period of 500 micro seconds (recommended). In timer IRQ handler, call this function.

Parameters

- instance – LPUART instance

void LIN_SetTimeoutCounter(uint32_t instance, uint32_t timeoutValue)

Set Value for Timeout Counter that is used in LIN_TimeoutService.

Parameters

- instance – LPUART instance
- timeoutValue – Timeout Value to be set

lin_status_t LIN_MasterSendHeader(uint32_t instance, uint8_t id)

Sends frame header out through the LIN Hardware Interface using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- instance – LPUART instance
- id – Frame Identifier

Returns

An error code or *lin_status_t*

lin_status_t LIN_EnableIRQ(uint32_t instance)

Enables LIN hardware interrupts.

Parameters

- instance – LPUART instance

Returns

An error code or *lin_status_t*

lin_status_t LIN_DisableIRQ(uint32_t instance)

Disables LIN hardware interrupts.

Parameters

- instance – LPUART instance

Returns

An error code or *lin_status_t*

void LIN_IRQHandler(uint8_t instance)

Interrupt handler for LIN Hardware Interface.

Parameters

- instance – LPUART instance

lin_status_t LIN_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns

lin_status_t

uint8_t LIN_ProcessParity(uint8_t PID, uint8_t typeAction)

Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID. This is not a public API as it is called by other API functions.

Parameters

- PID – PID byte in case of checking parity bits or ID byte in case of making parity bits.
- typeAction – 1 for Checking parity bits, 0 for making parity bits

Returns

0xFF if parity bits are incorrect, ID in case of checking parity bits and they are correct. Function returns PID in case of making parity bits.

uint8_t LIN_MakeChecksumByte(const uint8_t *buffer, uint8_t sizeBuffer, uint8_t PID)

Makes the checksum byte for a frame.

Parameters

- buffer – Pointer to Tx buffer
- sizeBuffer – Number of bytes that are contained in the buffer.
- PID – Protected Identifier byte.

Returns

the checksum byte.

FSL_LIN_DRIVER_VERSION

LIN driver version.

enum lin_frame_type

List of LIN frame types.

Values:

enumerator LIN_FRM_UNCD
unconditional frame

enumerator LIN_FRM_EVNT
event triggered frame

enumerator LIN_FRM_SPRDC
sporadic frame

enumerator LIN_FRM_DIAG
diagnostic frame

enum lin_frame_response

List of LIN frame response types.

Values:

enumerator LIN_RES_NOTHING
node is neither publisher nor subscriber

enumerator LIN_RES_PUB
node is publisher

enumerator LIN_RES_SUB
node is subscriber

enum lin_diagnostic_mode

Diagnostic mode.

Values:

enumerator DIAG_NONE
None

enumerator DIAG_INTERLEAVE_MODE
Interleave mode

enumerator DIAG_ONLY_MODE
Diagnostic only mode

enum lin_event_id_t

Defines types for an enumerating event related to an Identifier.

Values:

enumerator LIN_NO_EVENT
No event yet

enumerator LIN_WAKEUP_SIGNAL
Received a wakeup signal

enumerator LIN_BAUDRATE_ADJUSTED
Indicate that baudrate was adjusted to Master's baudrate

enumerator LIN_RECV_BREAK_FIELD_OK
Indicate that correct Break Field was received

enumerator LIN_SYNC_OK
Sync byte is correct

enumerator LIN_SYNC_ERROR
Sync byte is incorrect

enumerator LIN_PID_OK
PID correct

enumerator LIN_PID_ERROR
PID incorrect

enumerator LIN_FRAME_ERROR
Framing Error

enumerator LIN_READBACK_ERROR
Readback data is incorrect

enumerator LIN_CHECKSUM_ERROR
Checksum byte is incorrect

enumerator LIN_TX_COMPLETED
Sending data completed

enumerator LIN_RX_COMPLETED
Receiving data completed

enumerator LIN_NO_DATA_TIMEOUT
No data timeout

enumerator LIN_BUS_ACTIVITY_TIMEOUT
Bus activity timeout

enumerator LIN_TIMEOUT_ERROR
Indicate that timeout has occurred

enumerator LIN_LAST_RESPONSE_SHORT_ERROR
Indicate that the last frame was too short

enum lin_status_t

Defines Error codes of the LIN driver.

Values:

enumerator LIN_IFC_NOT_SUPPORT
This interface is not supported

enumerator LIN_INITIALIZED
LIN Hardware has been initialized

enumerator LIN_SUCCESS

Successfully done

enumerator LIN_ERROR

Error

enumerator LIN_TX_BUSY

Transmitter is busy

enumerator LIN_RX_BUSY

Receiver is busy

enumerator LIN_BUS_BUSY

Bus is busy

enumerator LIN_NO_TRANSFER_IN_PROGRESS

No data transfer is in progress

enumerator LIN_TIMEOUT

Timeout

enumerator LIN_LPUART_STAT_CLOCK_GATED_OFF

LPUART is gated from clock manager

enum lin_node_state_t

Define type for an enumerating LIN Node state.

Values:

enumerator LIN_NODE_STATE_UNINIT

Uninitialized state

enumerator LIN_NODE_STATE_SLEEP_MODE

Sleep mode state

enumerator LIN_NODE_STATE_IDLE

Idle state

enumerator LIN_NODE_STATE_SEND_BREAK_FIELD

Send break field state

enumerator LIN_NODE_STATE_RECV_SYNC

Receive the synchronization byte state

enumerator LIN_NODE_STATE_SEND_PID

Send PID state

enumerator LIN_NODE_STATE_RECV_PID

Receive PID state

enumerator LIN_NODE_STATE_RECV_DATA

Receive data state

enumerator LIN_NODE_STATE_RECV_DATA_COMPLETED

Receive data completed state

enumerator LIN_NODE_STATE_SEND_DATA

Send data state

enumerator LIN_NODE_STATE_SEND_DATA_COMPLETED

Send data completed state

enum lin_protocol_handle

List of protocols.

Values:

enumerator LIN_PROTOCOL_21

LIN protocol version 2.1

enumerator LIN_PROTOCOL_J2602

J2602 protocol

enum lin_supported_baudrates_t

List of supported baudrates for autobaud feature.

Values:

enumerator kLIN_BAUD_2400

enumerator kLIN_BAUD_4800

enumerator kLIN_BAUD_9600

enumerator kLIN_BAUD_14400

enumerator kLIN_BAUD_19200

typedef void (*lin_timer_get_time_interval_t)(uint32_t *nanoSeconds)

Callback function to get time interval in nano seconds.

typedef void (*lin_callback_t)(uint32_t instance, void *linState)

LIN Driver callback function type.

LPUART_Type *const g_linLpuartBase[1]

Table of base addresses for LPUART instances.

const IRQn_Type g_linLpuartRxTxIrqId[1]

Table to save LPUART IRQ enumeration numbers defined in the CMSIS header file.

lin_baudrate_values_t g_linConfigBaudrates[5U]

Table to save LIN user config buadrates values.

lin_state_t *g_linStatePtr[1]

Pointers to LPUART bases for each instance.

Table to save LPUART state structure pointers

lin_user_config_t *g_linUserconfigPtr[1]

Table to save LIN user config structure pointers.

LIN_SLAVE

LIN_MASTER

MAKE_PARITY

CHECK_PARITY

LIN_TIME_OUT_UNIT_US

LIN_MAKE_UNCONDITIONAL_FRAME

LIN_UPDATE_UNCONDITIONAL_FRAME

LIN_NUM_OF_SUPP_BAUDRATES

struct lin_user_config_t

#include <fsl_lin.h> LIN hardware configuration structure.

Public Members

`uint8_t hardware_instance`
interface instance number

`uint32_t baudRate`
baudrate of LIN Hardware Interface to configure

`bool nodeFunction`
Node function as Master or Slave

`bool autobaudEnable`
Enable Autobaud feature

`lin_timer_get_time_interval_t timerGetTimeIntervalCallback`
Callback function to get time interval in nano seconds

`struct lin_state_t`

#include <fsl_lin.h> Runtime state of the LIN driver.

Note that the caller provides memory for the driver state structures during initialization because the driver does not statically allocate memory.

Public Members

`const uint8_t *txBuff`
The buffer of data being sent.

`uint8_t *rxBuff`
The buffer of received data.

`uint8_t cntByte`
To count number of bytes already transmitted or received.

`volatile uint8_t txSize`
The remaining number of bytes to be received.

`volatile uint8_t rxSize`
The remaining number of bytes to be received.

`uint8_t checksum`
Checksum byte.

`volatile bool isTxBusy`
True if the LIN interface is transmitting frame data.

`volatile bool isRxBusy`
True if the LIN interface is receiving frame data.

`volatile bool isBusBusy`
True if there are data, frame headers being transferred on bus

`volatile bool isTxBlocking`
True if transmit is blocking transaction.

`volatile bool isRxBlocking`
True if receive is blocking transaction.

`lin_callback_t Callback`
Callback function to invoke after receiving a byte or transmitting a byte.

`uint8_t` `currentId`
Current ID

`uint8_t` `currentPid`
Current PID

volatile `lin_event_id_t` `currentEventId`
Current ID Event

volatile `lin_node_state_t` `currentNodeState`
Current Node state

volatile `uint32_t` `timeoutCounter`
Value of the timeout counter

volatile `bool` `timeoutCounterFlag`
Timeout counter flag

volatile `bool` `baudrateEvalEnable`
Baudrate Evaluation Process Enable

volatile `uint8_t` `fallingEdgeInterruptCount`
Falling Edge count of a sync byte

`uint32_t` `linSourceClockFreq`
Frequency of the source clock for LIN

volatile `bool` `txCompleted`
Used to wait for LIN interface ISR to complete transmission.

volatile `bool` `rxCompleted`
Used to wait for LIN interface ISR to complete reception

`struct` `lin_baudrate_values_t`
#include <fsl_lin.h> Structure of baudrate properties.

`struct` `lin_frame_struct`
#include <fsl_lin.h> Informations of frame.

Public Members

`lin_frame_type` `frm_type`
Frame information (unconditional or event triggered..)

`uint8_t` `frm_len`
Length of the frame

`lin_frame_response` `frm_response`
Action response when received PID

`uint8_t` `frm_offset`
Frame byte offset in frame buffer

`uint8_t` `flag_offset`
Flag byte offset in flag buffer

`uint8_t` `flag_size`
Flag size in flag buffer

`uint32_t` `delay`
Frame delay

```
const uint8_t *frame_data_ptr
```

List of Signal to which the frame is associated and its offset

```
struct lin_protocol_user_config_t
```

#include <fsl_lin.h> Protocol configuration structure.

Public Members

```
lin_protocol_handle protocol_version
```

Protocol version

```
lin_protocol_handle language_version
```

Language version

```
uint8_t number_of_configurable_frames
```

Number of frame except diagnostic frames

```
uint8_t frame_start
```

Start index of frame list

```
const lin_frame_struct *frame_tbl_ptr
```

Frame list except diagnostic frames

```
const uint16_t *list_identifiers_ROM_ptr
```

Configuration in ROM

```
uint8_t *list_identifiers_RAM_ptr
```

Configuration in RAM

```
uint16_t max_idle_timeout_cnt
```

Max Idle timeout counter

```
uint16_t max_message_length
```

Max message length

```
struct lin_product_id
```

#include <fsl_lin.h> Product id structure.

Public Members

```
uint16_t supplier_id
```

Supplier ID

```
uint16_t function_id
```

Function ID

```
uint8_t variant
```

Variant value

```
struct lin_serial_number
```

#include <fsl_lin.h> Serial number.

Public Members

```
uint8_t serial_0
```

Serial 0

```
uint8_t serial_1
```

Serial 1

uint8_t serial_2
Serial 2

uint8_t serial_3
Serial 3

struct lin_protocol_state_t
#include <fsl_lin.h> Protocol state structure.

Public Members

uint16_t baud_rate
Adjusted baud rate

uint8_t *response_buffer_ptr
Response buffer

uint8_t response_length
Response length

uint8_t successful_transfer
Sets when frame is transferred successfully

uint8_t error_in_response
Sets when frame received/transmitter by the node contains an error in the response field

uint8_t timeout_in_response
Timeout response

bool go_to_sleep_flg
Go to sleep flag

uint8_t current_id
Current PID

uint8_t num_of_processed_frame
Number of processed frames

uint8_t num_of_successfull_frame
Number of processed frames

uint8_t next_transmit_tick
Used to count the next transmit tick

bool save_config_flg
Set when save configuration request has been received

lin_diagnostic_mode diagnostic_mode
Diagnostic mode

uint16_t frame_timeout_cnt
Frame timeout counter

uint16_t idle_timeout_cnt
Idle timeout counter, node will go to sleep when count down to 0

bool transmit_error_resp_sig_flg
Flag indicates that the error response signal is going to be sent

bool event_trigger_collision_flg

Flag indicates collision on bus

struct lin_node_attribute

#include <fsl_lin.h> Attributes of LIN node.

Public Members

uint8_t *configured_NAD_ptr

NAD value used in configuration command

uint8_t initial_NAD

Initial NAD

lin_product_id product_id

Product ID

lin_serial_number serial_number

Serial number

uint8_t *resp_err_frm_id_ptr

Pointer to the list of index of frames that carries response error signal

uint8_t num_frame_have_esignal

The count of frames that carry response error signal

uint8_t response_error_byte_offset

Byte offset of response error signal

uint8_t response_error_bit_offset

Bit offset of response error signal

uint8_t num_of_fault_state_signal

Number of Fault state signal

uint16_t P2_min

P2 min

uint16_t ST_min

ST min

uint16_t N_As_timeout

N_As_timeout

uint16_t N_Cr_timeout

N_Cr_timeout

struct lin_master_data_t

#include <fsl_lin.h> LIN master data structure.

Public Members

uint8_t active_schedule_id

Active schedule table id

uint8_t previous_schedule_id

Previous schedule table id

uint8_t *schedule_start_entry_ptr

Start entry of each schedule table

```
uint8_t data_buffer[8]
    Master data buffer
```

2.38 LIN LPUART Driver

```
FSL_LIN_LPUART_DRIVER_VERSION
    LIN LPUART driver version.
```

```
enum _lin_lpuart_stop_bit_count
```

Values:

```
enumerator kLPUART_OneStopBit
```

One stop bit

```
enumerator kLPUART_TwoStopBit
```

Two stop bits

```
enum _lin_lpuart_flags
```

Values:

```
enumerator kLPUART_TxDataRegEmptyFlag
```

Transmit data register empty flag, sets when transmit buffer is empty

```
enumerator kLPUART_TransmissionCompleteFlag
```

Transmission complete flag, sets when transmission activity complete

```
enumerator kLPUART_RxDataRegFullFlag
```

Receive data register full flag, sets when the receive data buffer is full

```
enumerator kLPUART_IdleLineFlag
```

Idle line detect flag, sets when idle line detected

```
enumerator kLPUART_RxOverrunFlag
```

Receive Overrun, sets when new data is received before data is read from receive register

```
enumerator kLPUART_NoiseErrorFlag
```

Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets

```
enumerator kLPUART_FramingErrorFlag
```

Frame error flag, sets if logic 0 was detected where stop bit expected

```
enumerator kLPUART_ParityErrorFlag
```

If parity enabled, sets upon parity error detection

```
enumerator kLPUART_LinBreakFlag
```

LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

```
enumerator kLPUART_RxActiveEdgeFlag
```

Receive pin active edge interrupt flag, sets when active edge detected

```
enumerator kLPUART_RxActiveFlag
```

Receiver Active Flag (RAF), sets at beginning of valid start bit

```
enumerator kLPUART_DataMatch1Flag
```

The next character to be read from LPUART_DATA matches MA1

enumerator kLPUART_DataMatch2Flag

The next character to be read from LPUART_DATA matches MA2

enumerator kLPUART_NoiseErrorInRxDataRegFlag

NOISY bit, sets if noise detected in current data word

enumerator kLPUART_ParityErrorInRxDataRegFlag

PARITY bit, sets if noise detected in current data word

enumerator kLPUART_TxFifoEmptyFlag

TXEMPT bit, sets if transmit buffer is empty

enumerator kLPUART_RxFifoEmptyFlag

RXEMPT bit, sets if receive buffer is empty

enumerator kLPUART_TxFifoOverflowFlag

TXOF bit, sets if transmit buffer overflow occurred

enumerator kLPUART_RxFifoUnderflowFlag

RXUF bit, sets if receive buffer underflow occurred

enum _lin_lpuart_interrupt_enable

Values:

enumerator kLPUART_LinBreakInterruptEnable

LIN break detect.

enumerator kLPUART_RxActiveEdgeInterruptEnable

Receive Active Edge.

enumerator kLPUART_TxDataRegEmptyInterruptEnable

Transmit data register empty.

enumerator kLPUART_TransmissionCompleteInterruptEnable

Transmission complete.

enumerator kLPUART_RxDataRegFullInterruptEnable

Receiver data register full.

enumerator kLPUART_IdleLineInterruptEnable

Idle line.

enumerator kLPUART_RxOverrunInterruptEnable

Receiver Overrun.

enumerator kLPUART_NoiseErrorInterruptEnable

Noise error flag.

enumerator kLPUART_FramingErrorInterruptEnable

Framing error flag.

enumerator kLPUART_ParityErrorInterruptEnable

Parity error flag.

enumerator kLPUART_TxFifoOverflowInterruptEnable

Transmit FIFO Overflow.

enumerator kLPUART_RxFifoUnderflowInterruptEnable

Receive FIFO Underflow.

enum _lin_lpuart_status

Values:

```
enumerator kStatus_LPUART_TxBusy
    TX busy
enumerator kStatus_LPUART_RxBusy
    RX busy
enumerator kStatus_LPUART_TxIdle
    LPUART transmitter is idle.
enumerator kStatus_LPUART_RxIdle
    LPUART receiver is idle.
enumerator kStatus_LPUART_TxWatermarkTooLarge
    TX FIFO watermark too large
enumerator kStatus_LPUART_RxWatermarkTooLarge
    RX FIFO watermark too large
enumerator kStatus_LPUART_FlagCannotClearManually
    Some flag can't manually clear
enumerator kStatus_LPUART_Error
    Error happens on LPUART.
enumerator kStatus_LPUART_RxRingBufferOverrun
    LPUART RX software ring buffer overrun.
enumerator kStatus_LPUART_RxHardwareOverrun
    LPUART RX receiver overrun.
enumerator kStatus_LPUART_NoiseError
    LPUART noise error.
enumerator kStatus_LPUART_FramingError
    LPUART framing error.
enumerator kStatus_LPUART_ParityError
    LPUART parity error.
enum lin_lpuart_bit_count_per_char_t
    Values:
    enumerator LPUART_8_BITS_PER_CHAR
        8-bit data characters
    enumerator LPUART_9_BITS_PER_CHAR
        9-bit data characters
    enumerator LPUART_10_BITS_PER_CHAR
        10-bit data characters
typedef enum lin_lpuart_stop_bit_count lin_lpuart_stop_bit_count_t
static inline bool LIN_LPUART_GetRxDataPolarity(const LPUART_Type *base)
static inline void LIN_LPUART_SetRxDataPolarity(LPUART_Type *base, bool polarity)
static inline void LIN_LPUART_WriteByte(LPUART_Type *base, uint8_t data)
static inline void LIN_LPUART_ReadByte(const LPUART_Type *base, uint8_t *readData)
```

```
status_t LIN_LPUART_CalculateBaudRate(LPUART_Type *base, uint32_t baudRate_Bps,  
                                     uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)
```

Calculates the best osr and sbr value for configured baudrate.

Parameters

- base – LPUART peripheral base address
- baudRate_Bps – user configuration structure of type `lin_user_config_t`
- srcClock_Hz – pointer to the LIN_LPUART driver state structure
- osr – pointer to osr value
- sbr – pointer to sbr value

Returns

An error code or `lin_status_t`

```
void LIN_LPUART_SetBaudRate(LPUART_Type *base, uint32_t *osr, uint16_t *sbr)
```

Configure baudrate according to osr and sbr value.

Parameters

- base – LPUART peripheral base address
- osr – pointer to osr value
- sbr – pointer to sbr value

```
lin_status_t LIN_LPUART_Init(LPUART_Type *base, lin_user_config_t *linUserConfig,  
                             lin_state_t *linCurrentState, uint32_t linSourceClockFreq)
```

Initializes an LIN_LPUART instance for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN_LPUART clock source in the application to initialize the LIN_LPUART. This function initializes a LPUART instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, set break field length to be 13 bit times minimum, enable the break detect interrupt, Rx complete interrupt, frame error detect interrupt, and enable the LPUART module transmitter and receiver

Parameters

- base – LPUART peripheral base address
- linUserConfig – user configuration structure of type `lin_user_config_t`
- linCurrentState – pointer to the LIN_LPUART driver state structure
- linSourceClockFreq – LIN source clock frequency in Hz

Returns

An error code or `lin_status_t`

```
lin_status_t LIN_LPUART_Deinit(LPUART_Type *base)
```

Shuts down the LIN_LPUART by disabling interrupts and transmitter/receiver.

Parameters

- base – LPUART peripheral base address

Returns

An error code or `lin_status_t`

```
lin_status_t LIN_LPUART_SendFrameDataBlocking(LPUART_Type *base, const uint8_t *txBuff,  
                                              uint8_t txSize, uint32_t timeoutMSec)
```

Sends Frame data out through the LIN_LPUART module using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_SendFrameData(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN_LPUART module using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_GetTransmitStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. This function return LIN_TX_BUSY while sending, or LIN_TIMEOUT if timeout has occurred, or return LIN_SUCCESS when the transmission is complete. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to transmit

Returns

`lin_status_t` LIN_TX_BUSY, LIN_SUCCESS or LIN_TIMEOUT

lin_status_t LIN_LPUART_RecvFrmDataBlocking(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN_LPUART module using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_RecvFrmData(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize)

Receives frame data through the LIN_LPUART module using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_AbortTransferData(LPUART_Type *base)

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- base – LPUART peripheral base address

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_GetReceiveStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking reception While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function return the current event ID, LIN_RX_BUSY while receiving and return LIN_SUCCESS, or 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

2.39 LPADC: 12-bit SAR Analog-to-Digital Converter Driver

enum _lpadc_status_flags

Define hardware flags of the module.

Values:

enumerator kLPADC_ResultFIFO0OverflowFlag

Indicates that more data has been written to the Result FIFO 0 than it can hold.

enumerator kLPADC_ResultFIFO0ReadyFlag

Indicates when the number of valid datawords in the result FIFO 0 is greater than the setting watermark level.

enumerator kLPADC_TriggerExceptionFlag

Indicates that a trigger exception event has occurred.

enumerator kLPADC_TriggerCompletionFlag

Indicates that a trigger completion event has occurred.

enumerator kLPADC_CalibrationReadyFlag

Indicates that the calibration process is done.

enumerator kLPADC_ActiveFlag

Indicates that the ADC is in active state.

enumerator kLPADC_ResultFIFOOverflowFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowFlag as instead.

enumerator kLPADC_ResultFIFOReadyFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0ReadyFlag as instead.

enum _lpadc_interrupt_enable

Define interrupt switchers of the module.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_ResultFIFO0OverflowInterruptEnable

Configures ADC to generate overflow interrupt requests when FOF0 flag is asserted.

enumerator kLPADC_FIFO0WatermarkInterruptEnable

Configures ADC to generate watermark interrupt requests when RDY0 flag is asserted.

enumerator kLPADC_ResultFIFOOverflowInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowInterruptEnable as instead.

enumerator kLPADC_FIFOWatermarkInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_FIFO0WatermarkInterruptEnable as instead.

enumerator kLPADC_TriggerExceptionInterruptEnable

Configures ADC to generate trigger exception interrupt.

enumerator kLPADC_Trigger0CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 0 completion.

enumerator kLPADC_Trigger1CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 1 completion.

enumerator kLPADC_Trigger2CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 2 completion.

enumerator kLPADC_Trigger3CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 3 completion.

enumerator kLPADC_Trigger4CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 4 completion.

enumerator kLPADC_Trigger5CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 5 completion.

enumerator kLPADC_Trigger6CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 6 completion.

enumerator kLPADC_Trigger7CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 7 completion.

enumerator kLPADC_Trigger8CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 8 completion.

enumerator kLPADC_Trigger9CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 9 completion.

enumerator kLPADC_Trigger10CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 10 completion.

enumerator kLPADC_Trigger11CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 11 completion.

enumerator kLPADC_Trigger12CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 12 completion.

enumerator kLPADC_Trigger13CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 13 completion.

enumerator kLPADC_Trigger14CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 14 completion.

enumerator kLPADC_Trigger15CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 15 completion.

enum _lpadc_trigger_status_flags

The enumerator of lpadc trigger status flags, including interrupted flags and completed flags.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_Trigger0InterruptedFlag
Trigger 0 is interrupted by a high priority exception.

enumerator kLPADC_Trigger1InterruptedFlag
Trigger 1 is interrupted by a high priority exception.

enumerator kLPADC_Trigger2InterruptedFlag
Trigger 2 is interrupted by a high priority exception.

enumerator kLPADC_Trigger3InterruptedFlag
Trigger 3 is interrupted by a high priority exception.

enumerator kLPADC_Trigger4InterruptedFlag
Trigger 4 is interrupted by a high priority exception.

enumerator kLPADC_Trigger5InterruptedFlag
Trigger 5 is interrupted by a high priority exception.

enumerator kLPADC_Trigger6InterruptedFlag
Trigger 6 is interrupted by a high priority exception.

enumerator kLPADC_Trigger7InterruptedFlag
Trigger 7 is interrupted by a high priority exception.

enumerator kLPADC_Trigger8InterruptedFlag
Trigger 8 is interrupted by a high priority exception.

enumerator kLPADC_Trigger9InterruptedFlag

Trigger 9 is interrupted by a high priority exception.

enumerator kLPADC_Trigger10InterruptedFlag

Trigger 10 is interrupted by a high priority exception.

enumerator kLPADC_Trigger11InterruptedFlag

Trigger 11 is interrupted by a high priority exception.

enumerator kLPADC_Trigger12InterruptedFlag

Trigger 12 is interrupted by a high priority exception.

enumerator kLPADC_Trigger13InterruptedFlag

Trigger 13 is interrupted by a high priority exception.

enumerator kLPADC_Trigger14InterruptedFlag

Trigger 14 is interrupted by a high priority exception.

enumerator kLPADC_Trigger15InterruptedFlag

Trigger 15 is interrupted by a high priority exception.

enumerator kLPADC_Trigger0CompletedFlag

Trigger 0 is completed and trigger 0 has enabled completion interrupts.

enumerator kLPADC_Trigger1CompletedFlag

Trigger 1 is completed and trigger 1 has enabled completion interrupts.

enumerator kLPADC_Trigger2CompletedFlag

Trigger 2 is completed and trigger 2 has enabled completion interrupts.

enumerator kLPADC_Trigger3CompletedFlag

Trigger 3 is completed and trigger 3 has enabled completion interrupts.

enumerator kLPADC_Trigger4CompletedFlag

Trigger 4 is completed and trigger 4 has enabled completion interrupts.

enumerator kLPADC_Trigger5CompletedFlag

Trigger 5 is completed and trigger 5 has enabled completion interrupts.

enumerator kLPADC_Trigger6CompletedFlag

Trigger 6 is completed and trigger 6 has enabled completion interrupts.

enumerator kLPADC_Trigger7CompletedFlag

Trigger 7 is completed and trigger 7 has enabled completion interrupts.

enumerator kLPADC_Trigger8CompletedFlag

Trigger 8 is completed and trigger 8 has enabled completion interrupts.

enumerator kLPADC_Trigger9CompletedFlag

Trigger 9 is completed and trigger 9 has enabled completion interrupts.

enumerator kLPADC_Trigger10CompletedFlag

Trigger 10 is completed and trigger 10 has enabled completion interrupts.

enumerator kLPADC_Trigger11CompletedFlag

Trigger 11 is completed and trigger 11 has enabled completion interrupts.

enumerator kLPADC_Trigger12CompletedFlag

Trigger 12 is completed and trigger 12 has enabled completion interrupts.

enumerator kLPADC_Trigger13CompletedFlag

Trigger 13 is completed and trigger 13 has enabled completion interrupts.

enumerator kLPADC_Trigger14CompletedFlag

Trigger 14 is completed and trigger 14 has enabled completion interrupts.

enumerator kLPADC_Trigger15CompletedFlag

Trigger 15 is completed and trigger 15 has enabled completion interrupts.

enum _lpadc_sample_scale_mode

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Values:

enumerator kLPADC_SamplePartScale

Use divided input voltage signal. (For scale select, please refer to the reference manual).

enumerator kLPADC_SampleFullScale

Full scale (Factor of 1).

enum _lpadc_sample_channel_mode

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Values:

enumerator kLPADC_SampleChannelSingleEndSideA

Single-end mode, only A-side channel is converted.

enumerator kLPADC_SampleChannelSingleEndSideB

Single-end mode, only B-side channel is converted.

enumerator kLPADC_SampleChannelDiffBothSideAB

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDiffBothSideBA

Differential mode, the ADC result is (CHnB-CHnA).

enumerator kLPADC_SampleChannelDiffBothSide

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDualSingleEndBothSide

Dual-Single-Ended Mode. Both A side and B side channels are converted independently.

enum _lpadc_hardware_average_mode

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

Values:

enumerator kLPADC__HardwareAverageCount1
Single conversion.

enumerator kLPADC__HardwareAverageCount2
2 conversions averaged.

enumerator kLPADC__HardwareAverageCount4
4 conversions averaged.

enumerator kLPADC__HardwareAverageCount8
8 conversions averaged.

enumerator kLPADC__HardwareAverageCount16
16 conversions averaged.

enumerator kLPADC__HardwareAverageCount32
32 conversions averaged.

enumerator kLPADC__HardwareAverageCount64
64 conversions averaged.

enumerator kLPADC__HardwareAverageCount128
128 conversions averaged.

enum _lpadc_sample_time_mode

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Values:

enumerator kLPADC__SampleTimeADCK3
3 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK5
5 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK7
7 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK11
11 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK19
19 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK35
35 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK67
69 ADCK cycles total sample time.

enumerator kLPADC__SampleTimeADCK131
131 ADCK cycles total sample time.

enum _lpadc_hardware_compare_mode

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally

only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Values:

enumerator kLPADC__HardwareCompareDisabled

Compare disabled.

enumerator kLPADC__HardwareCompareStoreOnTrue

Compare enabled. Store on true.

enumerator kLPADC__HardwareCompareRepeatUntilTrue

Compare enabled. Repeat channel acquisition until true.

enum _lpadc_conversion_resolution_mode

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

Values:

enumerator kLPADC__ConversionResolutionStandard

Standard resolution. Single-ended 12-bit conversion, Differential 13-bit conversion with 2's complement output.

enumerator kLPADC__ConversionResolutionHigh

High resolution. Single-ended 16-bit conversion; Differential 16-bit conversion with 2's complement output.

enum _lpadc_conversion_average_mode

Define enumeration of conversion averages mode.

Configure the conversion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

Values:

enumerator kLPADC__ConversionAverage1

Single conversion.

enumerator kLPADC__ConversionAverage2

2 conversions averaged.

enumerator kLPADC__ConversionAverage4

4 conversions averaged.

enumerator kLPADC__ConversionAverage8

8 conversions averaged.

enumerator kLPADC__ConversionAverage16

16 conversions averaged.

enumerator kLPADC__ConversionAverage32

32 conversions averaged.

enumerator kLPADC__ConversionAverage64

64 conversions averaged.

enumerator kLPADC__ConversionAverage128

128 conversions averaged.

enumerator kLPADC_ConversionAverageMax

enum __lpadc_reference_voltage_mode

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

Values:

enumerator kLPADC_ReferenceVoltageAlt1

Option 1 setting.

enumerator kLPADC_ReferenceVoltageAlt2

Option 2 setting.

enumerator kLPADC_ReferenceVoltageAlt3

Option 3 setting.

enum __lpadc_power_level_mode

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Values:

enumerator kLPADC_PowerLevelAlt1

Lowest power setting.

enumerator kLPADC_PowerLevelAlt2

Next lowest power setting.

enumerator kLPADC_PowerLevelAlt3

...

enumerator kLPADC_PowerLevelAlt4

Highest power setting.

enum __lpadc_offset_calibration_mode

Define enumeration of offset calibration mode.

Values:

enumerator kLPADC_OffsetCalibration12bitMode

12 bit offset calibration mode.

enumerator kLPADC_OffsetCalibration16bitMode

16 bit offset calibration mode.

enum __lpadc_trigger_priority_policy

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

Values:

enumerator kLPADC_ConvPreemptImmediatelyNotAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion is not automatically resumed or restarted.

enumerator kLPADC_ConvPreemptSoftlyNotAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion is not resumed or restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoRestarted

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptSoftlyAutoRestarted

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be resumed.

enumerator kLPADC_ConvPreemptSoftlyAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will be automatically be resumed.

enumerator kLPADC_TriggerPriorityPreemptImmediately

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityPreemptSoftly

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityExceptionDisabled

High priority trigger exception disabled.

enum _lpadc_tune_value

Define enumeration of tune value.

Values:

enumerator kLPADC_TuneValue0

Tune value 0.

enumerator kLPADC_TuneValue1

Tune value 1.

enumerator kLPADC_TuneValue2

Tune value 2.

enumerator kLPADC_TuneValue3

Tune value 3.

typedef enum *_lpadc_sample_scale_mode* lpadc_sample_scale_mode_t

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

typedef enum *_lpadc_sample_channel_mode* lpadc_sample_channel_mode_t

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

typedef enum *_lpadc_hardware_average_mode* lpadc_hardware_average_mode_t

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

typedef enum *_lpadc_sample_time_mode* lpadc_sample_time_mode_t

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

typedef enum *_lpadc_hardware_compare_mode* lpadc_hardware_compare_mode_t

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

typedef enum *_lpadc_conversion_resolution_mode* lpadc_conversion_resolution_mode_t

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

typedef enum *_lpadc_conversion_average_mode* lpadc_conversion_average_mode_t

Define enumeration of conversion averages mode.

Configure the converion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

`typedef enum _lpadc_reference_voltage_mode` `lpadc_reference_voltage_source_t`

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

`typedef enum _lpadc_power_level_mode` `lpadc_power_level_mode_t`

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

`typedef enum _lpadc_offset_calibration_mode` `lpadc_offset_calibration_mode_t`

Define enumeration of offset calibration mode.

`typedef enum _lpadc_trigger_priority_policy` `lpadc_trigger_priority_policy_t`

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

`typedef enum _lpadc_tune_value` `lpadc_tune_value_t`

Define enumeration of tune value.

`typedef struct _lpadc_calibration_value` `lpadc_calibration_value_t`

A structure of calibration value.

`LPADC_CONVERSION_COMPLETE_TIMEOUT`

Max loops to wait for LPADC conversion complete.

When doing calibration, driver will wait for the completion of conversion. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

`LPADC_CALIBRATION_READY_TIMEOUT`

Max loops to wait for LPADC calibration ready.

Before doing calibration, driver will wait for the calibration ready. This parameter defines how many loops to check the calibration ready. If defined as 0, driver will wait forever until ready.

`LPADC_GAIN_CAL_READY_TIMEOUT`

Max loops to wait for LPADC gain calibration GAIN_CAL ready.

Before doing calibration, driver will wait for the gain calibration GAIN_CAL ready. This parameter defines how many loops to check the gain calibration GAIN_CAL ready. If defined as 0, driver will wait forever until ready.

`ADC_OFSTRIM_OFSTRIM_MAX`

`ADC_OFSTRIM_OFSTRIM_SIGN`

`LPADC_GET_ACTIVE_COMMAND_STATUS(statusVal)`

Define the MACRO function to get command status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

`LPADC_GET_ACTIVE_TRIGGER_STATUE(statusVal)`

Define the MACRO function to get trigger status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

`void LPADC_Init(ADC_Type *base, const lpadc_config_t *config)`

Initializes the LPADC module.

Parameters

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

`void LPADC_GetDefaultConfig(lpadc_config_t *config)`

Gets an available pre-defined settings for initial configuration.

This function initializes the converter configuration structure with an available settings. The default values are:

```
config->enableInDozeMode      = true;
config->enableAnalogPreliminary = false;
config->powerUpDelay           = 0x80;
config->referenceVoltageSource  = kLPADC_ReferenceVoltageAlt1;
config->powerLevelMode         = kLPADC_PowerLevelAlt1;
config->triggerPriorityPolicy    = kLPADC_TriggerPriorityPreemptImmediately;
config->enableConvPause        = false;
config->convPauseDelay          = 0U;
config->FIFOWatermark           = 0U;
```

Parameters

- `config` – Pointer to configuration structure.

`void LPADC_Deinit(ADC_Type *base)`

De-initializes the LPADC module.

Parameters

- `base` – LPADC peripheral base address.

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

Switch on/off the LPADC module.

Parameters

- `base` – LPADC peripheral base address.
- `enable` – switcher to the module.

`static inline void LPADC_DoResetFIFO(ADC_Type *base)`

Do reset the conversion FIFO.

Parameters

- `base` – LPADC peripheral base address.

`static inline void LPADC_DoResetConfig(ADC_Type *base)`

Do reset the module's configuration.

Reset all ADC internal logic and registers, except the Control Register (ADCx_CTRL).

Parameters

- `base` – LPADC peripheral base address.

`static inline uint32_t LPADC_GetStatusFlags(ADC_Type *base)`

Get status flags.

Parameters

- `base` – LPADC peripheral base address.

Returns

status flags' mask. See to `_lpadc_status_flags`.

static inline void LPADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clear status flags.

Only the flags can be cleared by writing ADCx_STATUS register would be cleared by this API.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for flags to be cleared. See to `_lpadc_status_flags`.

static inline uint32_t LPADC_GetTriggerStatusFlags(ADC_Type *base)

Get trigger status flags to indicate which trigger sequences have been completed or interrupted by a high priority trigger exception.

Parameters

- base – LPADC peripheral base address.

Returns

The OR'ed value of `_lpadc_trigger_status_flags`.

static inline void LPADC_ClearTriggerStatusFlags(ADC_Type *base, uint32_t mask)

Clear trigger status flags.

Parameters

- base – LPADC peripheral base address.
- mask – The mask of trigger status flags to be cleared, should be the OR'ed value of `_lpadc_trigger_status_flags`.

static inline void LPADC_EnableInterrupts(ADC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

static inline void LPADC_DisableInterrupts(ADC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

static inline void LPADC_EnableFIFOWatermarkDMA(ADC_Type *base, bool enable)

Switch on/off the DMA trigger for FIFO watermark event.

Parameters

- base – LPADC peripheral base address.
- enable – Switcher to the event.

static inline uint32_t LPADC_GetConvResultCount(ADC_Type *base)

Get the count of result kept in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

Returns

The count of result kept in conversion FIFO.

```
bool LPADC_GetConvResult(ADC_Type *base, lpadc_conv_result_t *result)
```

Get the result in conversion FIFO.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

Returns

Status whether FIFO entry is valid.

```
void LPADC_GetConvResultBlocking(ADC_Type *base, lpadc_conv_result_t *result)
```

Get the result in conversion FIFO using blocking method.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

```
void LPADC_SetConvTriggerConfig(ADC_Type *base, uint32_t triggerId, const  
                                lpadc_conv_trigger_config_t *config)
```

Configure the conversion trigger source.

Each programmable trigger can launch the conversion command in command buffer.

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- config – Pointer to configuration structure. See to `lpadc_conv_trigger_config_t`.

```
void LPADC_GetDefaultConvTriggerConfig(lpadc_conv_trigger_config_t *config)
```

Gets an available pre-defined settings for trigger's configuration.

This function initializes the trigger's configuration structure with an available settings. The default values are:

```
config->targetCommandId    = 0U;  
config->delayPower         = 0U;  
config->priority           = 0U;  
config->channelAFIFOSelect = 0U;  
config->channelBFIFOSelect = 0U;  
config->enableHardwareTrigger = false;
```

Parameters

- config – Pointer to configuration structure.

```
static inline void LPADC_DoSoftwareTrigger(ADC_Type *base, uint32_t triggerIdMask)
```

Do software trigger to conversion command.

Parameters

- base – LPADC peripheral base address.
- triggerIdMask – Mask value for software trigger indexes, which count from zero.

```
static inline void LPADC_EnableHardwareTriggerCommandSelection(ADC_Type *base, uint32_t
                                                                triggerId, bool enable)
```

Enable hardware trigger command selection.

This function will use the hardware trigger command from ADC_ETC. The trigger command is then defined by ADC hardware trigger command selection field in ADC_ETC->TRIGx_CHAINy_z_n[CSEL].

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- enable – True to enable or false to disable.

```
void LPADC_SetConvCommandConfig(ADC_Type *base, uint32_t commandId, const
                                lpadc_conv_command_config_t *config)
```

Configure conversion command.

Note: The number of compare value register on different chips is different, that is mean in some chips, some command buffers do not have the compare functionality.

Parameters

- base – LPADC peripheral base address.
- commandId – ID for command in command buffer. Typically, the available value range is 1 - 15.
- config – Pointer to configuration structure. See to lpadc_conv_command_config_t.

```
void LPADC_GetDefaultConvCommandConfig(lpadc_conv_command_config_t *config)
```

Gets an available pre-defined settings for conversion command's configuration.

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
config->sampleScaleMode      = kLPADC_SampleFullScale;
config->channelBScaleMode    = kLPADC_SampleFullScale;
config->sampleChannelMode     = kLPADC_SampleChannelSingleEndSideA;
config->channelNumber         = 0U;
config->channelBNumber        = 0U;
config->chainedNextCommandNumber = 0U;
config->enableAutoChannelIncrement = false;
config->loopCount              = 0U;
config->hardwareAverageMode    = kLPADC_HardwareAverageCount1;
config->sampleTimeMode        = kLPADC_SampleTimeADCK3;
config->hardwareCompareMode    = kLPADC_HardwareCompareDisabled;
config->hardwareCompareValueHigh = 0U;
config->hardwareCompareValueLow  = 0U;
config->conversionResolutionMode = kLPADC_ConversionResolutionStandard;
config->enableWaitTrigger       = false;
config->enableChannelB          = false;
```

Parameters

- config – Pointer to configuration structure.

`void LPADC_EnableCalibration(ADC_Type *base, bool enable)`

Enable the calibration function.

When CALOFS is set, the ADC is configured to perform a calibration function anytime the ADC executes a conversion. Any channel selected is ignored and the value returned in the RESFIFO is a signed value between -31 and 31. -32 is not a valid and is never a returned value. Software should copy the lower 6-bits of the conversion result stored in the RESFIFO after a completed calibration conversion to the OFSTRIM field. The OFSTRIM field is used in normal operation for offset correction.

Parameters

- `base` – LPADC peripheral base address.
- `enable` – switcher to the calibration function.

`static inline void LPADC_SetOffsetValue(ADC_Type *base, uint32_t value)`

Set proper offset value to trim ADC.

To minimize the offset during normal operation, software should read the conversion result from the RESFIFO calibration operation and write the lower 6 bits to the OFSTRIM register.

Parameters

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

`status_t LPADC_DoAutoCalibration(ADC_Type *base)`

Do auto calibration.

Calibration function should be executed before using converter in application. It used the software trigger and a dummy conversion, get the offset and write them into the OFSTRIM register. It called some of functional API including:

- `LPADC_EnableCalibration(...)`
- `LPADC_SetOffsetValue(...)`
- `LPADC_SetConvCommandConfig(...)`
- `LPADC_SetConvTriggerConfig(...)`

Parameters

- `base` – LPADC peripheral base address.
- `base` – LPADC peripheral base address.

Return values

- `kStatus_Success` – Successfully configured.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

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

Set trim value for offset.

Note: For 16-bit conversions, each increment is 1/2 LSB resulting in a programmable offset range of -256 LSB to 255.5 LSB; For 12-bit conversions, each increment is 1/32 LSB resulting in a programmable offset range of -16 LSB to 15.96875 LSB.

Parameters

- `base` – LPADC peripheral base address.
- `value` – Offset trim value, is a 10-bit signed value between -512 and 511.

```
static inline void LPADC_GetOffsetValue(ADC_Type *base, int16_t *pValue)
```

Get trim value of offset.

Parameters

- base – LPADC peripheral base address.
- pValue – Pointer to the variable in type of int16_t to store offset value.

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

Enable the offset calibration function.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

```
static inline void LPADC_SetOffsetCalibrationMode(ADC_Type *base,  
                                                  lpadc_offset_calibration_mode_t mode)
```

Set offset calibration mode.

Parameters

- base – LPADC peripheral base address.
- mode – set offset calibration mode.see to lpadc_offset_calibration_mode_t.

```
status_t LPADC_DoOffsetCalibration(ADC_Type *base)
```

Do offset calibration.

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
void LPADC_PrepareAutoCalibration(ADC_Type *base)
```

Prepare auto calibration, LPADC_FinishAutoCalibration has to be called before using the LPADC. LPADC_DoAutoCalibration has been split in two API to avoid to be stuck too long in the function.

Parameters

- base – LPADC peripheral base address.

```
status_t LPADC_FinishAutoCalibration(ADC_Type *base)
```

Finish auto calibration start with LPADC_PrepareAutoCalibration.

Note: This feature is used for LPADC with CTRL[CALOFSMODE].

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
void LPADC_GetCalibrationValue(ADC_Type *base, lpadc_calibration_value_t  
                             *ptrCalibrationValue)
```

Get calibration value into the memory which is defined by invoker.

Note: Please note the ADC will be disabled temporary.

Note: This function should be used after finish calibration.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure, this memory block should be always powered on even in low power modes.

```
status_t LPADC_SetCalibrationValue(ADC_Type *base, const lpadc_calibration_value_t  
                                  *ptrCalibrationValue)
```

Set calibration value into ADC calibration registers.

Note: Please note the ADC will be disabled temporary.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure which contains ADC's calibration value.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
static inline void LPADC_RequestHighSpeedModeTrim(ADC_Type *base)
```

Request high speed mode trim calculation.

Parameters

- base – LPADC peripheral base address.

```
static inline int8_t LPADC_GetHighSpeedTrimValue(ADC_Type *base)
```

Get high speed mode trim value, the result is a 5-bit signed value between -16 and 15.

Note: The high speed mode trim value is used to minimize offset for high speed conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The calculated high speed mode trim value.

```
static inline void LPADC_SetHighSpeedTrimValue(ADC_Type *base, int8_t trimValue)
```

Set high speed mode trim value.

Note: If is possible to set the trim value manually, but it is recommended to use the LPADC_RequestHighSpeedModeTrim.

Parameters

- base – LPADC peripheral base address.
- trimValue – The trim value to be set.

static inline void LPADC_EnableHighSpeedConversionMode(ADC_Type *base, bool enable)

Enable/disable high speed conversion mode, if enabled conversions complete 2 or 3 ADCK cycles sooner compared to conversion cycle counts when high speed mode is disabled.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable high speed conversion mode:
 - **true** Enable high speed conversion mode;
 - **false** Disable high speed conversion mode.

static inline void LPADC_EnableExtraCycle(ADC_Type *base, bool enable)

Enable/disable an additional ADCK cycle to conversion.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable an additional ADCK cycle to conversion:
 - **true** Enable an additional ADCK cycle to conversion;
 - **false** Disable an additional ADCK cycle to conversion.

static inline void LPADC_SetTuneValue(ADC_Type *base, *lpadc_tune_value_t* tuneValue)

Set tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.
- tuneValue – The tune value to be set, please refer to *lpadc_tune_value_t*.

static inline *lpadc_tune_value_t* LPADC_GetTuneValue(ADC_Type *base)

Get tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The tune value, please refer to *lpadc_tune_value_t*.

FSL_LPADC_DRIVER_VERSION

LPADC driver version 2.9.5.

struct *lpadc_config_t*

#include <fsl_lpadc.h> LPADC global configuration.

This structure would used to keep the settings for initialization.

Public Members

bool enableInternalClock

Enables the internally generated clock source. The clock source is used in clock selection logic at the chip level and is optionally used for the ADC clock source.

bool enableVref1LowVoltage

If voltage reference option1 input is below 1.8V, it should be “true”. If voltage reference option1 input is above 1.8V, it should be “false”.

bool enableInDozeMode

Control system transition to Stop and Wait power modes while ADC is converting. When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

lpadc_conversion_average_mode_t conversionAverageMode

Auto-Calibration Averages.

bool enableAnalogPreliminary

ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).

uint32_t powerUpDelay

When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize. The startup delay count of (powerUpDelay * 4) ADCK cycles must result in a longer delay than the analog startup time.

lpadc_reference_voltage_source_t referenceVoltageSource

Selects the voltage reference high used for conversions.

lpadc_power_level_mode_t powerLevelMode

Power Configuration Selection.

lpadc_trigger_priority_policy_t triggerPriorityPolicy

Control how higher priority triggers are handled, see to *lpadc_trigger_priority_policy_t*.

bool enableConvPause

Enables the ADC pausing function. When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in “Compare Until True” configuration.

uint32_t convPauseDelay

Controls the duration of pausing during command execution sequencing. The pause delay is a count of (convPauseDelay*4) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

uint32_t FIFOWatermark

FIFOWatermark is a programmable threshold setting. When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

struct lpadc_conv_command_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion command.

Public Members

lpadc_sample_scale_mode_t sampleScaleMode

Sample scale mode.

lpadc_sample_scale_mode_t channelBScaleMode

Alternate channel B Scale mode.

lpadc_sample_channel_mode_t sampleChannelMode

Channel sample mode.

uint32_t channelNumber

Channel number; select the channel or channel pair.

uint32_t channelBNumber

Alternate Channel B number; select the channel.

uint32_t chainedNextCommandNumber

Selects the next command to be executed after this command completes. 1-15 is available, 0 is to terminate the chain after this command.

bool enableAutoChannelIncrement

Loop with increment: when disabled, the “loopCount” field selects the number of times the selected channel is converted consecutively; when enabled, the “loopCount” field defines how many consecutive channels are converted as part of the command execution.

uint32_t loopCount

Selects how many times this command executes before finish and transition to the next command or Idle state. Command executes LOOP+1 times. 0-15 is available.

lpadc_hardware_average_mode_t hardwareAverageMode

Hardware average selection.

lpadc_sample_time_mode_t sampleTimeMode

Sample time selection.

lpadc_hardware_compare_mode_t hardwareCompareMode

Hardware compare selection.

uint32_t hardwareCompareValueHigh

Compare Value High. The available value range is in 16-bit.

uint32_t hardwareCompareValueLow

Compare Value Low. The available value range is in 16-bit.

lpadc_conversion_resolution_mode_t conversionResolutionMode

Conversion resolution mode.

bool enableWaitTrigger

Wait for trigger assertion before execution: when disabled, this command will be automatically executed; when enabled, the active trigger must be asserted again before executing this command.

struct *lpadc_conv_trigger_config_t*

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion trigger.

Public Members

uint32_t targetCommandId

Select the command from command buffer to execute upon detect of the associated trigger event.

uint32_t delayPower

Select the trigger delay duration to wait at the start of servicing a trigger event. When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is $2^{\text{delayPower}}$ ADCK cycles. The available value range is 4-bit.

uint32_t priority

Sets the priority of the associated trigger source. If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

bool enableHardwareTrigger

Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not. The software trigger is always available.

struct lpadc_conv_result_t

#include <fsl_lpadc.h> Define the structure to keep the conversion result.

Public Members

uint32_t commandIdSource

Indicate the command buffer being executed that generated this result.

uint32_t loopCountIndex

Indicate the loop count value during command execution that generated this result.

uint32_t triggerIdSource

Indicate the trigger source that initiated a conversion and generated this result.

uint16_t convValue

Data result.

struct __lpadc_calibration_value

#include <fsl_lpadc.h> A structure of calibration value.

2.40 LPCMP: Low Power Analog Comparator Driver

void LPCMP_Init(LPCMP_Type *base, const *lpcmp_config_t* *config)

Initialize the LPCMP.

This function initializes the LPCMP module. The operations included are:

- Enabling the clock for LPCMP module.
- Configuring the comparator.
- Enabling the LPCMP module. Note: For some devices, multiple LPCMP instance share the same clock gate. In this case, to enable the clock for any instance enables all the LPCMPs. Check the chip reference manual for the clock assignment of the LPCMP.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to “lpcmp_config_t” structure.

void LPCMP_Deinit(LPCMP_Type *base)

De-initializes the LPCMP module.

This function de-initializes the LPCMP module. The operations included are:

- Disabling the LPCMP module.
- Disabling the clock for LPCMP module.

This function disables the clock for the LPCMP. Note: For some devices, multiple LPCMP instance shares the same clock gate. In this case, before disabling the clock for the LPCMP, ensure that all the LPCMP instances are not used.

Parameters

- base – LPCMP peripheral base address.

void LPCMP_GetDefaultConfig(*lpcmp_config_t* *config)

Gets an available pre-defined settings for the comparator's configuration.

This function initializes the comparator configuration structure to these default values:

```
config->enableStopMode    = false;
config->enableOutputPin    = false;
config->enableCmpToDacLink = false;
config->useUnfilteredOutput = false;
config->enableInvertOutput  = false;
config->hysteresisMode      = kLPCMP_HysteresisLevel0;
config->powerMode           = kLPCMP_LowSpeedPowerMode;
config->functionalSourceClock = kLPCMP_FunctionalClockSource0;
config->plusInputSrc        = kLPCMP_PlusInputSrcMux;
config->minusInputSrc       = kLPCMP_MinusInputSrcMux;
```

Parameters

- config – Pointer to “lpcmp_config_t” structure.

static inline void LPCMP_Enable(LPCMP_Type *base, bool enable)

Enable/Disable LPCMP module.

Parameters

- base – LPCMP peripheral base address.
- enable – “true” means enable the module, and “false” means disable the module.

void LPCMP_SetInputChannels(LPCMP_Type *base, uint32_t positiveChannel, uint32_t negativeChannel)

Select the input channels for LPCMP. This function determines which input is selected for the negative and positive mux.

Parameters

- base – LPCMP peripheral base address.
- positiveChannel – Positive side input channel number. Available range is 0-7.
- negativeChannel – Negative side input channel number. Available range is 0-7.

static inline void LPCMP_EnableDMA(LPCMP_Type *base, bool enable)

Enables/disables the DMA request for rising/falling events. Normally, the LPCMP generates a CPU interrupt if there is a rising/falling event. When DMA support is enabled and the rising/falling interrupt is enabled, the rising/falling event forces a DMA transfer request rather than a CPU interrupt instead.

Parameters

- base – LPCMP peripheral base address.
- enable – “true” means enable DMA support, and “false” means disable DMA support.

void LPCMP_SetFilterConfig(LPCMP_Type *base, const *lpcmp_filter_config_t* *config)

Configures the filter.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to “lpcmp_filter_config_t” structure.

void LPCMP_SetDACConfig(LPCMP_Type *base, const *lpcmp_dac_config_t* *config)

Configure the internal DAC module.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to “lpcmp_dac_config_t” structure. If config is “NULL”, disable internal DAC.

static inline void LPCMP_EnableInterrupts(LPCMP_Type *base, uint32_t mask)

Enable the interrupts.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for interrupts. See “_lpcmp_interrupt_enable”.

static inline void LPCMP_DisableInterrupts(LPCMP_Type *base, uint32_t mask)

Disable the interrupts.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for interrupts. See “_lpcmp_interrupt_enable”.

static inline uint32_t LPCMP_GetStatusFlags(LPCMP_Type *base)

Get the LPCMP status flags.

Parameters

- base – LPCMP peripheral base address.

Returns

Mask value for the asserted flags. See “_lpcmp_status_flags”.

static inline void LPCMP_ClearStatusFlags(LPCMP_Type *base, uint32_t mask)

Clear the LPCMP status flags.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for the flags. See “_lpcmp_status_flags”.

static inline void LPCMP_EnableWindowMode(LPCMP_Type *base, bool enable)

Enable/Disable window mode. When any windowed mode is active, COUTA is clocked by the bus clock whenever WINDOW = 1. The last latched value is held when WINDOW = 0. The optionally inverted comparator output COUT_RAW is sampled on every bus clock when WINDOW=1 to generate COUTA.

Parameters

- base – LPCMP peripheral base address.
- enable – “true” means enable window mode, and “false” means disable window mode.

```
void LPCMP_SetWindowControl(LPCMP_Type *base, const lpcmp_window_control_config_t *config)
```

Configure the window control, users can use this API to implement operations on the window, such as inverting the window signal, setting the window closing event(only valid in windowing mode), and setting the COUTA signal after the window is closed(only valid in windowing mode).

Parameters

- base – LPCMP peripheral base address.
- config – Pointer “lpcmp_window_control_config_t” structure.

FSL_LPCMP_DRIVER_VERSION

LPCMP driver version 2.3.2.

enum *_lpcmp_status_flags*

LPCMP status falgs mask.

Values:

enumerator kLPCMP_OutputRisingEventFlag

Rising-edge on the comparison output has occurred.

enumerator kLPCMP_OutputFallingEventFlag

Falling-edge on the comparison output has occurred.

enumerator kLPCMP_OutputAssertEventFlag

Return the current value of the analog comparator output. The flag does not support W1C.

enum *_lpcmp_interrupt_enable*

LPCMP interrupt enable/disable mask.

Values:

enumerator kLPCMP_OutputRisingInterruptEnable

Comparator interrupt enable rising.

enumerator kLPCMP_OutputFallingInterruptEnable

Comparator interrupt enable falling.

enum *_lpcmp_hysteresis_mode*

LPCMP hysteresis mode. See chip data sheet to get the actual hystersis value with each level.

Values:

enumerator kLPCMP_HysteresisLevel0

The hard block output has level 0 hysteresis internally.

enumerator kLPCMP_HysteresisLevel1

The hard block output has level 1 hysteresis internally.

enumerator kLPCMP_HysteresisLevel2

The hard block output has level 2 hysteresis internally.

enumerator kLPCMP_HysteresisLevel3

The hard block output has level 3 hysteresis internally.

enum `_lpcmp_power_mode`

LPCMP nano mode.

Values:

enumerator `kLPCMP_LowSpeedPowerMode`

Low speed comparison mode is selected.

enumerator `kLPCMP_HighSpeedPowerMode`

High speed comparison mode is selected.

enumerator `kLPCMP_NanoPowerMode`

Nano power comparator is enabled.

enum `_lpcmp_dac_reference_voltage_source`

Internal DAC reference voltage source.

Values:

enumerator `kLPCMP_VrefSourceVin1`

`vrefh_int` is selected as resistor ladder network supply reference Vin.

enumerator `kLPCMP_VrefSourceVin2`

`vrefh_ext` is selected as resistor ladder network supply reference Vin.

enum `_lpcmp_couta_signal`

Set the COUTA signal value when the window is closed.

Values:

enumerator `kLPCMP_COUTASignalNoSet`

NO set the COUTA signal value when the window is closed.

enumerator `kLPCMP_COUTASignalLow`

Set COUTA signal low(0) when the window is closed.

enumerator `kLPCMP_COUTASignalHigh`

Set COUTA signal high(1) when the window is closed.

enum `_lpcmp_close_window_event`

Set COUT event, which can close the active window in window mode.

Values:

enumerator `kLPCMP_CloseWindowEventNoSet`

No Set COUT event, which can close the active window in window mode.

enumerator `kLPCMP_CloseWindowEventRisingEdge`

Set rising edge COUT signal as COUT event.

enumerator `kLPCMP_CloseWindowEventFallingEdge`

Set falling edge COUT signal as COUT event.

enumerator `kLPCMP_CloseWindowEventBothEdge`

Set both rising and falling edge COUT signal as COUT event.

typedef enum `_lpcmp_hysteresis_mode` `lpcmp_hysteresis_mode_t`

LPCMP hysteresis mode. See chip data sheet to get the actual hysteresis value with each level.

typedef enum `_lpcmp_power_mode` `lpcmp_power_mode_t`

LPCMP nano mode.

typedef enum *_lpcmp_dac_reference_voltage_source* lpcmp_dac_reference_voltage_source_t
Internal DAC reference voltage source.

typedef enum *_lpcmp_couta_signal* lpcmp_couta_signal_t
Set the COUTA signal value when the window is closed.

typedef enum *_lpcmp_close_window_event* lpcmp_close_window_event_t
Set COUT event, which can close the active window in window mode.

typedef struct *_lpcmp_filter_config* lpcmp_filter_config_t
Configure the filter.

typedef struct *_lpcmp_dac_config* lpcmp_dac_config_t
configure the internal DAC.

typedef struct *_lpcmp_config* lpcmp_config_t
Configures the comparator.

typedef struct *_lpcmp_window_control_config* lpcmp_window_control_config_t
Configure the window mode control.

LPCMP_CCR1_COUTA_CFG_MASK

LPCMP_CCR1_COUTA_CFG_SHIFT

LPCMP_CCR1_COUTA_CFG(x)

LPCMP_CCR1_EVT_SEL_CFG_MASK

LPCMP_CCR1_EVT_SEL_CFG_SHIFT

LPCMP_CCR1_EVT_SEL_CFG(x)

struct *_lpcmp_filter_config*
#include <fsl_lpcmp.h> Configure the filter.

Public Members

bool enableSample
Decide whether to use the external SAMPLE as a sampling clock input.

uint8_t filterSampleCount
Filter Sample Count. Available range is 1-7; 0 disables the filter.

uint8_t filterSamplePeriod
Filter Sample Period. The divider to the bus clock. Available range is 0-255. The sampling clock must be at least 4 times slower than the system clock to the comparator. So if enableSample is “false”, filterSamplePeriod should be set greater than 4.

struct *_lpcmp_dac_config*
#include <fsl_lpcmp.h> configure the internal DAC.

Public Members

bool enableLowPowerMode
Decide whether to enable DAC low power mode.

lpcmp_dac_reference_voltage_source_t referenceVoltageSource
Internal DAC supply voltage reference source.

uint8_t DACValue

Value for the DAC Output Voltage. Different devices has different available range, for specific values, please refer to the reference manual.

struct __lpcmp_config

#include <fsl_lpcmp.h> Configures the comparator.

Public Members

bool enableOutputPin

Decide whether to enable the comparator is available in selected pin.

bool useUnfilteredOutput

Decide whether to use unfiltered output.

bool enableInvertOutput

Decide whether to inverts the comparator output.

lpcmp_hysteresis_mode_t hysteresisMode

LPCMP hysteresis mode.

lpcmp_power_mode_t powerMode

LPCMP power mode.

struct __lpcmp_window_control_config

#include <fsl_lpcmp.h> Configure the window mode control.

Public Members

bool enableInvertWindowSignal

True: enable invert window signal, False: disable invert window signal.

lpcmp_couta_signal_t COUTASignal

Decide whether to define the COUTA signal value when the window is closed.

lpcmp_close_window_event_t closeWindowEvent

Decide whether to select COUT event signal edge defines a COUT event to close window.

2.41 LPI2C: Low Power Inter-Integrated Circuit Driver

void LPI2C_DriverIRQHandler(uint32_t instance)

LPI2C driver IRQ handler common entry.

This function provides the common IRQ request entry for LPI2C.

Parameters

- instance – LPI2C instance.

FSL_LPI2C_DRIVER_VERSION

LPI2C driver version.

LPI2C status return codes.

Values:

enumerator kStatus_LPI2C_Busy

The master is already performing a transfer.

enumerator kStatus_LPI2C_Idle

The slave driver is idle.

enumerator kStatus_LPI2C_Nak

The slave device sent a NAK in response to a byte.

enumerator kStatus_LPI2C_FifoError

FIFO under run or overrun.

enumerator kStatus_LPI2C_BitError

Transferred bit was not seen on the bus.

enumerator kStatus_LPI2C_ArbitrationLost

Arbitration lost error.

enumerator kStatus_LPI2C_PinLowTimeout

SCL or SDA were held low longer than the timeout.

enumerator kStatus_LPI2C_NoTransferInProgress

Attempt to abort a transfer when one is not in progress.

enumerator kStatus_LPI2C_DmaRequestFail

DMA request failed.

enumerator kStatus_LPI2C_Timeout

Timeout polling status flags.

IRQn_Type const kLpi2cMasterIrqs[]

Array to map LPI2C instance number to IRQ number, used internally for LPI2C master interrupt and EDMA transactional APIs.

IRQn_Type const kLpi2cSlaveIrqs[]

lpi2c_master_isr_t s_lpi2cMasterIsr

Pointer to master IRQ handler for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

void *s_lpi2cMasterHandle[]

Pointers to master handles for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

uint32_t LPI2C_GetInstance(LPI2C_Type *base)

Returns an instance number given a base address.

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

- base – The LPI2C peripheral base address.

Returns

LPI2C instance number starting from 0.

I2C_RETRY_TIMES

Retry times for waiting flag.

2.42 LPI2C Master Driver

`void LPI2C_MasterGetDefaultConfig(lpi2c_master_config_t *masterConfig)`

Provides a default configuration for the LPI2C master peripheral.

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->debugEnable       = false;
masterConfig->ignoreAck         = false;
masterConfig->pinConfig          = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz       = 100000U;
masterConfig->busIdleTimeout_ns = 0;
masterConfig->pinLowTimeout_ns  = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable = false;
masterConfig->hostRequest.source  = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `LPI2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `lpi2c_master_config_t`.

`void LPI2C_MasterInit(LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t sourceClock_Hz)`

Initializes the LPI2C master peripheral.

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `LPI2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`void LPI2C_MasterDeinit(LPI2C_Type *base)`

Deinitializes the LPI2C master peripheral.

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

`void LPI2C_MasterConfigureDataMatch(LPI2C_Type *base, const lpi2c_data_match_config_t *matchConfig)`

Configures LPI2C master data match feature.

Parameters

- `base` – The LPI2C peripheral base address.
- `matchConfig` – Settings for the data match feature.

status_t LPI2C_MasterCheckAndClearError(LPI2C_Type *base, uint32_t status)

Convert provided flags to status code, and clear any errors if present.

Parameters

- base – The LPI2C peripheral base address.
- status – Current status flags value that will be checked.

Return values

- kStatus_Success –
- kStatus_LPI2C_PinLowTimeout –
- kStatus_LPI2C_ArbitrationLost –
- kStatus_LPI2C_Nak –
- kStatus_LPI2C_FifoError –

status_t LPI2C_CheckForBusyBus(LPI2C_Type *base)

Make sure the bus isn't already busy.

A busy bus is allowed if we are the one driving it.

Parameters

- base – The LPI2C peripheral base address.

Return values

- kStatus_Success –
- kStatus_LPI2C_Busy –

static inline void LPI2C_MasterReset(LPI2C_Type *base)

Performs a software reset.

Restores the LPI2C master peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_MasterEnable(LPI2C_Type *base, bool enable)

Enables or disables the LPI2C module as master.

Parameters

- base – The LPI2C peripheral base address.
- enable – Pass true to enable or false to disable the specified LPI2C as master.

static inline uint32_t LPI2C_MasterGetStatusFlags(LPI2C_Type *base)

Gets the LPI2C master status flags.

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_lpi2c_master_flags`

Parameters

- base – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void LPI2C_MasterClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)

Clears the LPI2C master status flag state.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

See also:

`_lpi2c_master_flags`.

Parameters

- base – The LPI2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_MasterGetStatusFlags()`.

static inline void LPI2C_MasterEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Enables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline void LPI2C_MasterDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Disables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t LPI2C_MasterGetEnabledInterrupts(LPI2C_Type *base)

Returns the set of currently enabled LPI2C master interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline void LPI2C_MasterEnableDMA(LPI2C_Type *base, bool enableTx, bool enableRx)
```

Enables or disables LPI2C master DMA requests.

Parameters

- `base` – The LPI2C peripheral base address.
- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.

```
static inline uint32_t LPI2C_MasterGetTxFifoAddress(LPI2C_Type *base)
```

Gets LPI2C master transmit data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

```
static inline uint32_t LPI2C_MasterGetRxFifoAddress(LPI2C_Type *base)
```

Gets LPI2C master receive data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

```
static inline void LPI2C_MasterSetWatermarks(LPI2C_Type *base, size_t txWords, size_t rxWords)
```

Sets the watermarks for LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txWords` – Transmit FIFO watermark value in words. The `kLPI2C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO is equal or less than `txWords`. Writing a value equal or greater than the FIFO size is truncated.
- `rxWords` – Receive FIFO watermark value in words. The `kLPI2C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO is greater than `rxWords`. Writing a value equal or greater than the FIFO size is truncated.

```
static inline void LPI2C_MasterGetFifoCounts(LPI2C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of words in the LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.

```
void LPI2C_MasterSetBaudRate(LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t
                             baudRate_Hz)
```

Sets the I2C bus frequency for master transactions.

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Note: Please note that the second parameter is the clock frequency of LPI2C module, the third parameter means user configured bus baudrate, this implementation is different from other I2C drivers which use baudrate configuration as second parameter and source clock frequency as third parameter.

Parameters

- `base` – The LPI2C peripheral base address.
- `sourceClock_Hz` – LPI2C functional clock frequency in Hertz.
- `baudRate_Hz` – Requested bus frequency in Hertz.

```
static inline bool LPI2C_MasterGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

```
status_t LPI2C_MasterStart(LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
```

Sends a START signal and slave address on the I2C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

```
static inline status_t LPI2C_MasterRepeatedStart(LPI2C_Type *base, uint8_t address,
                                                  lpi2c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `LPI2C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

status_t LPI2C_MasterSend(LPI2C_Type *base, void *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_LPI2C_Nak`.

Parameters

- `base` – The LPI2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or over run.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The LPI2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.

- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterStop(LPI2C_Type *base)`

Sends a STOP signal on the I2C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterTransferBlocking(LPI2C_Type *base, lpi2c_master_transfer_t *transfer)`

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- `base` – The LPI2C peripheral base address.
- `transfer` – Pointer to the transfer structure.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`void LPI2C_MasterTransferCreateHandle(LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)`

Creates a new handle for the LPI2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_MasterTransferNonBlocking(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *lpi2c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_LPI2C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t LPI2C_MasterTransferGetCount(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *size_t* *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void LPI2C_MasterTransferAbort(LPI2C_Type *base, *lpi2c_master_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

void LPI2C_MasterTransferHandleIRQ(LPI2C_Type *base, void *lpi2cMasterHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The LPI2C peripheral base address.
- lpi2cMasterHandle – Pointer to the LPI2C master driver handle.

enum _lpi2c_master_flags

LPI2C master peripheral flags.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kLPI2C_MasterTxReadyFlag

Transmit data flag

enumerator kLPI2C_MasterRxReadyFlag

Receive data flag

enumerator kLPI2C_MasterEndOfPacketFlag

End Packet flag

enumerator kLPI2C_MasterStopDetectFlag

Stop detect flag

enumerator kLPI2C_MasterNackDetectFlag

NACK detect flag

enumerator kLPI2C_MasterArbitrationLostFlag

Arbitration lost flag

enumerator kLPI2C_MasterFifoErrFlag

FIFO error flag

enumerator kLPI2C_MasterPinLowTimeoutFlag

Pin low timeout flag

enumerator kLPI2C_MasterDataMatchFlag

Data match flag

enumerator kLPI2C_MasterBusyFlag

Master busy flag

enumerator kLPI2C_MasterBusBusyFlag

Bus busy flag

enumerator kLPI2C_MasterClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_MasterIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_MasterErrorFlags

Errors to check for.

enum _lpi2c_direction

Direction of master and slave transfers.

Values:

enumerator kLPI2C_Write

Master transmit.

enumerator kLPI2C_Read

Master receive.

enum _lpi2c_master_pin_config

LPI2C pin configuration.

Values:

enumerator kLPI2C_2PinOpenDrain

LPI2C Configured for 2-pin open drain mode

enumerator kLPI2C_2PinOutputOnly

LPI2C Configured for 2-pin output only mode (ultra-fast mode)

enumerator kLPI2C_2PinPushPull

LPI2C Configured for 2-pin push-pull mode

enumerator kLPI2C_4PinPushPull

LPI2C Configured for 4-pin push-pull mode

enumerator kLPI2C_2PinOpenDrainWithSeparateSlave

LPI2C Configured for 2-pin open drain mode with separate LPI2C slave

enumerator kLPI2C_2PinOutputOnlyWithSeparateSlave

LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave

enumerator kLPI2C_2PinPushPullWithSeparateSlave

LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave

enumerator kLPI2C_4PinPushPullWithInvertedOutput

LPI2C Configured for 4-pin push-pull mode(inverted outputs)

enum _lpi2c_host_request_source

LPI2C master host request selection.

Values:

enumerator kLPI2C_HostRequestExternalPin

Select the LPI2C_HREQ pin as the host request input

enumerator kLPI2C_HostRequestInputTrigger

Select the input trigger as the host request input

enum _lpi2c_host_request_polarity

LPI2C master host request pin polarity configuration.

Values:

enumerator kLPI2C_HostRequestPinActiveLow

Configure the LPI2C_HREQ pin active low

enumerator kLPI2C_HostRequestPinActiveHigh

Configure the LPI2C_HREQ pin active high

enum _lpi2c_data_match_config_mode

LPI2C master data match configuration modes.

Values:

enumerator kLPI2C_MatchDisabled

LPI2C Match Disabled

enumerator kLPI2C_1stWordEqualsM0OrM1

LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1

enumerator kLPI2C_AnyWordEqualsM0OrM1

LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1

enumerator kLPI2C_1stWordEqualsM0And2ndWordEqualsM1

LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1

enumerator kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1

LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1

enumerator kLPI2C_1stWordAndM1EqualsM0AndM1

LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1

enumerator kLPI2C_AnyWordAndM1EqualsM0AndM1

LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1

enum _lpi2c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_lpi2c_master_transfer::flags` field.

Values:

enumerator kLPI2C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kLPI2C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kLPI2C_TransferRepeatedStartFlag

Send a repeated start condition

enumerator kLPI2C_TransferNoStopFlag

Don't send a stop condition.

```
typedef enum _lpi2c_direction lpi2c_direction_t
```

Direction of master and slave transfers.

```
typedef enum _lpi2c_master_pin_config lpi2c_master_pin_config_t
```

LPI2C pin configuration.

```
typedef enum _lpi2c_host_request_source lpi2c_host_request_source_t
```

LPI2C master host request selection.

```
typedef enum _lpi2c_host_request_polarity lpi2c_host_request_polarity_t
```

LPI2C master host request pin polarity configuration.

```
typedef struct _lpi2c_master_config lpi2c_master_config_t
```

Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _lpi2c_data_match_config_mode lpi2c_data_match_config_mode_t
```

LPI2C master data match configuration modes.

```
typedef struct _lpi2c_match_config lpi2c_data_match_config_t
```

LPI2C master data match configuration structure.

```
typedef struct _lpi2c_master_transfer lpi2c_master_transfer_t
```

LPI2C master descriptor of the transfer.

```
typedef struct _lpi2c_master_handle lpi2c_master_handle_t
```

LPI2C master handle of the transfer.

```
typedef void (*lpi2c_master_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterTransferCreateHandle().

Param base

The LPI2C peripheral base address.

Param handle

Pointer to the LPI2C master driver handle.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef void (*lpi2c_master_isr_t)(LPI2C_Type *base, void *handle)
```

Typedef for master interrupt handler, used internally for LPI2C master interrupt and EDMA transactional APIs.

```
struct _lpi2c_master_config
```

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

bool enableDoze

Whether master is enabled in doze mode.

bool debugEnable

Enable transfers to continue when halted in debug mode.

bool ignoreAck

Whether to ignore ACK/NACK.

lpi2c_master_pin_config_t pinConfig

The pin configuration option.

uint32_t baudRate_Hz

Desired baud rate in Hertz.

uint32_t busIdleTimeout_ns

Bus idle timeout in nanoseconds. Set to 0 to disable.

uint32_t pinLowTimeout_ns

Pin low timeout in nanoseconds. Set to 0 to disable.

uint8_t sdaGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SDA pin. Set to 0 to disable.

uint8_t sclGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SCL pin. Set to 0 to disable.

struct *_lpi2c_master_config* hostRequest

Host request options.

struct *_lpi2c_match_config*

#include <fsl_lpi2c.h> LPI2C master data match configuration structure.

Public Members

lpi2c_data_match_config_mode_t matchMode

Data match configuration setting.

bool rxDataMatchOnly

When set to true, received data is ignored until a successful match.

uint32_t match0

Match value 0.

uint32_t match1

Match value 1.

struct *_lpi2c_master_transfer*

#include <fsl_lpi2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_lpi2c_master_transfer_flags` for available options. Set to 0 or `kLPI2C_TransferDefaultFlag` for normal transfers.

uint16_t slaveAddress

The 7-bit slave address.

lpi2c_direction_t direction

Either `kLPI2C_Read` or `kLPI2C_Write`.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct `_lpi2c_master_handle`

#include <fsl_lpi2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint16_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

uint16_t commandBuffer[6]

LPI2C command sequence. When all 6 command words are used: `Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word]`

lpi2c_master_transfer_t transfer

Copy of the current transfer info.

lpi2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData

Application data passed to callback.

struct hostRequest

Public Members

`bool enable`

Enable host request.

`lpi2c_host_request_source_t source`

Host request source.

`lpi2c_host_request_polarity_t polarity`

Host request pin polarity.

2.43 LPI2C Master DMA Driver

```
void LPI2C_MasterCreateEDMAHandle(LPI2C_Type *base, lpi2c_master_edma_handle_t *handle,
                                   edma_handle_t *rxDmaHandle, edma_handle_t
                                   *txDmaHandle, lpi2c_master_edma_transfer_callback_t
                                   callback, void *userData)
```

Create a new handle for the LPI2C master DMA APIs.

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_MasterTransferAbortEDMA()` API shall be called.

For devices where the LPI2C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – **[out]** Pointer to the LPI2C master driver handle.
- *rxDmaHandle* – Handle for the eDMA receive channel. Created by the user prior to calling this function.
- *txDmaHandle* – Handle for the eDMA transmit channel. Created by the user prior to calling this function.
- *callback* – User provided pointer to the asynchronous callback function.
- *userData* – User provided pointer to the application callback data.

```
status_t LPI2C_MasterTransferEDMA(LPI2C_Type *base, lpi2c_master_edma_handle_t *handle,
                                   lpi2c_master_transfer_t *transfer)
```

Performs a non-blocking DMA-based transaction on the I2C bus.

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to the LPI2C master driver handle.
- *transfer* – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_LPI2C_Busy` – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

status_t LPI2C_MasterTransferGetCountEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a DMA transaction currently in progress.

status_t LPI2C_MasterTransferAbortEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the eDMA peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

Return values

- kStatus_Success – A transaction was successfully aborted.
- kStatus_LPI2C_Idle – There is not a DMA transaction currently in progress.

typedef struct *lpi2c_master_edma_handle* *lpi2c_master_edma_handle_t*
LPI2C master EDMA handle of the transfer.

typedef void (**lpi2c_master_edma_transfer_callback_t*)(LPI2C_Type *base,
lpi2c_master_edma_handle_t *handle, *status_t* completionStatus, void *userData)

Master DMA completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterCreateEDMAHandle().

Param base

The LPI2C peripheral base address.

Param handle

Handle associated with the completed transfer.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
struct _lpi2c_master_edma_handle
    #include <fsl_lpi2c_edma.h> Driver handle for master DMA APIs.
```

Note: The contents of this structure are private and subject to change.

Public Members

LPI2C_Type *base
LPI2C base pointer.

bool isBusy
Transfer state machine current state.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

uint16_t commandBuffer[20]
LPI2C command sequence. When all 10 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word] + receive&Size[4 words]

lpi2c_master_transfer_t transfer
Copy of the current transfer info.

lpi2c_master_edma_transfer_callback_t completionCallback
Callback function pointer.

void *userData
Application data passed to callback.

edma_handle_t *rx
Handle for receive DMA channel.

edma_handle_t *tx
Handle for transmit DMA channel.

edma_tcd_t tcds[3]
Software TCD. Three are allocated to provide enough room to align to 32-bytes.

2.44 LPI2C Slave Driver

```
void LPI2C_SlaveGetDefaultConfig(lpi2c_slave_config_t *slaveConfig)
```

Provides a default configuration for the LPI2C slave peripheral.

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave      = true;
slaveConfig->address0         = 0U;
slaveConfig->address1         = 0U;
slaveConfig->addressMatchMode = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable = true;
slaveConfig->filterEnable     = true;
slaveConfig->enableGeneralCall = false;
slaveConfig->sclStall.enableAck = false;
slaveConfig->sclStall.enableTx  = true;
slaveConfig->sclStall.enableRx  = true;
```

(continues on next page)

(continued from previous page)

```
slaveConfig->sclStall.enableAddress    = true;
slaveConfig->ignoreAck                 = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns   = 0;
slaveConfig->sclGlitchFilterWidth_ns   = 0;
slaveConfig->dataValidDelay_ns         = 0;
slaveConfig->clockHoldTime_ns          = 0;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `LPI2C_SlaveInit()`. Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure that is set to default values. Refer to `lpi2c_slave_config_t`.

```
void LPI2C_SlaveInit(LPI2C_Type *base, const lpi2c_slave_config_t *slaveConfig, uint32_t
                    sourceClock_Hz)
```

Initializes the LPI2C slave peripheral.

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `LPI2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

```
void LPI2C_SlaveDeinit(LPI2C_Type *base)
```

Deinitializes the LPI2C slave peripheral.

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveReset(LPI2C_Type *base)
```

Performs a software reset of the LPI2C slave peripheral.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveEnable(LPI2C_Type *base, bool enable)
```

Enables or disables the LPI2C module as slave.

Parameters

- `base` – The LPI2C peripheral base address.
- `enable` – Pass true to enable or false to disable the specified LPI2C as slave.

```
static inline uint32_t LPI2C_SlaveGetStatusFlags(LPI2C_Type *base)
```

Gets the LPI2C slave status flags.

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_lpi2c_slave_flags`

Parameters

- `base` – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void LPI2C_SlaveClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)
```

Clears the LPI2C status flag state.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

Attempts to clear other flags has no effect.

See also:

`_lpi2c_slave_flags`.

Parameters

- `base` – The LPI2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_SlaveGetStatusFlags()`.

```
static inline void LPI2C_SlaveEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Enables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void LPI2C_SlaveDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Disables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t LPI2C_SlaveGetEnabledInterrupts(LPI2C_Type *base)
```

Returns the set of currently enabled LPI2C slave interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline void LPI2C_SlaveEnableDMA(LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)
```

Enables or disables the LPI2C slave peripheral DMA requests.

Parameters

- base – The LPI2C peripheral base address.
- enableAddressValid – Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
- enableRx – Enable flag for the receive data DMA request. Pass true for enable, false for disable.
- enableTx – Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

```
static inline bool LPI2C_SlaveGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the slave mode to be enabled.

Parameters

- base – The LPI2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

```
static inline void LPI2C_SlaveTransmitAck(LPI2C_Type *base, bool ackOrNack)
```

Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.

Use this function to send an ACK or NAK when the `kLPI2C_SlaveTransmitAckFlag` is asserted. This only happens if you enable the `sclStall.enableAck` field of the `lpi2c_slave_config_t` configuration structure used to initialize the slave peripheral.

Parameters

- base – The LPI2C peripheral base address.
- ackOrNack – Pass true for an ACK or false for a NAK.

```
static inline void LPI2C_SlaveEnableAckStall(LPI2C_Type *base, bool enable)
```

Enables or disables ACKSTALL.

When enables ACKSTALL, software can transmit either an ACK or NAK on the I2C bus in response to a byte from the master.

Parameters

- base – The LPI2C peripheral base address.
- enable – True will enable ACKSTALL, false will disable ACKSTALL.

```
static inline uint32_t LPI2C_SlaveGetReceivedAddress(LPI2C_Type *base)
```

Returns the slave address sent by the I2C master.

This function should only be called if the `kLPI2C_SlaveAddressValidFlag` is asserted.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

```
status_t LPI2C_SlaveSend(LPI2C_Type *base, void *txBuff, size_t txSize, size_t *actualTxSize)
```

Performs a polling send transfer on the I2C bus.

Parameters

- `base` – The LPI2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.
- `actualTxSize` – **[out]**

Returns

Error or success status returned by API.

```
status_t LPI2C_SlaveReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)
```

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The LPI2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.
- `actualRxSize` – **[out]**

Returns

Error or success status returned by API.

```
void LPI2C_SlaveTransferCreateHandle(LPI2C_Type *base, lpi2c_slave_handle_t *handle,  
                                     lpi2c_slave_transfer_callback_t callback, void *userData)
```

Creates a new handle for the LPI2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_SlaveTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The LPI2C peripheral base address.
- `handle` – **[out]** Pointer to the LPI2C slave driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

```
status_t LPI2C_SlaveTransferNonBlocking(LPI2C_Type *base, lpi2c_slave_handle_t *handle,
                                         uint32_t eventMask)
```

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of *lpi2c_slave_transfer_event_t* enumerators for the events you wish to receive. The *kLPI2C_SlaveTransmitEvent* and *kLPI2C_SlaveReceiveEvent* events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the *kLPI2C_SlaveAllEvents* constant is provided as a convenient way to enable all events.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *lpi2c_slave_handle_t* structure which stores the transfer state.
- *eventMask* – Bit mask formed by OR'ing together *lpi2c_slave_transfer_event_t* enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and *kLPI2C_SlaveAllEvents* to enable all events.

Return values

- *kStatus_Success* – Slave transfers were successfully started.
- *kStatus_LPI2C_Busy* – Slave transfers have already been started on this handle.

```
status_t LPI2C_SlaveTransferGetCount(LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t
                                     *count)
```

Gets the slave transfer status during a non-blocking transfer.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *i2c_slave_handle_t* structure.
- *count* – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- *kStatus_Success* –
- *kStatus_NoTransferInProgress* –

```
void LPI2C_SlaveTransferAbort(LPI2C_Type *base, lpi2c_slave_handle_t *handle)
```

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- *base* – The LPI2C peripheral base address.

- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`void LPI2C_SlaveTransferHandleIRQ(LPI2C_Type *base, lpi2c_slave_handle_t *handle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The LPI2C peripheral base address.
- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`enum _lpi2c_slave_flags`

LPI2C slave peripheral flags.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator `kLPI2C_SlaveTxReadyFlag`

Transmit data flag

enumerator `kLPI2C_SlaveRxReadyFlag`

Receive data flag

enumerator `kLPI2C_SlaveAddressValidFlag`

Address valid flag

enumerator `kLPI2C_SlaveTransmitAckFlag`

Transmit ACK flag

enumerator `kLPI2C_SlaveRepeatedStartDetectFlag`

Repeated start detect flag

enumerator `kLPI2C_SlaveStopDetectFlag`

Stop detect flag

enumerator `kLPI2C_SlaveBitErrFlag`

Bit error flag

enumerator `kLPI2C_SlaveFifoErrFlag`

FIFO error flag

enumerator `kLPI2C_SlaveAddressMatch0Flag`

Address match 0 flag

enumerator kLPI2C_SlaveAddressMatch1Flag

Address match 1 flag

enumerator kLPI2C_SlaveGeneralCallFlag

General call flag

enumerator kLPI2C_SlaveBusyFlag

Master busy flag

enumerator kLPI2C_SlaveBusBusyFlag

Bus busy flag

enumerator kLPI2C_SlaveClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_SlaveIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_SlaveErrorFlags

Errors to check for.

enum _lpi2c_slave_address_match

LPI2C slave address match options.

Values:

enumerator kLPI2C_MatchAddress0

Match only address 0.

enumerator kLPI2C_MatchAddress0OrAddress1

Match either address 0 or address 1.

enumerator kLPI2C_MatchAddress0ThroughAddress1

Match a range of slave addresses from address 0 through address 1.

enum _lpi2c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kLPI2C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kLPI2C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kLPI2C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kLPI2C_SlaveTransmitAckEvent

Callback needs to either transmit an ACK or NACK.

enumerator kLPI2C_SlaveRepeatedStartEvent

A repeated start was detected.

enumerator kLPI2C_SlaveCompletionEvent

A stop was detected, completing the transfer.

enumerator kLPI2C_SlaveAllEvents

Bit mask of all available events.

typedef enum *_lpi2c_slave_address_match* lpi2c_slave_address_match_t

LPI2C slave address match options.

typedef struct *_lpi2c_slave_config* lpi2c_slave_config_t

Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_lpi2c_slave_transfer_event* lpi2c_slave_transfer_event_t

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct *_lpi2c_slave_transfer* lpi2c_slave_transfer_t

LPI2C slave transfer structure.

typedef struct *_lpi2c_slave_handle* lpi2c_slave_handle_t

LPI2C slave handle structure.

typedef void (*lpi2c_slave_transfer_callback_t)(LPI2C_Type *base, *lpi2c_slave_transfer_t* *transfer, void *userData)

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the LPI2C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

struct *_lpi2c_slave_config*

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

`bool enableSlave`
Enable slave mode.

`uint8_t address0`
Slave's 7-bit address.

`uint8_t address1`
Alternate slave 7-bit address.

`lpi2c_slave_address_match_t addressMatchMode`
Address matching options.

`bool filterDozeEnable`
Enable digital glitch filter in doze mode.

`bool filterEnable`
Enable digital glitch filter.

`bool enableGeneralCall`
Enable general call address matching.

`struct _lpi2c_slave_config sclStall`
SCL stall enable options.

`bool ignoreAck`
Continue transfers after a NACK is detected.

`bool enableReceivedAddressRead`
Enable reading the address received address as the first byte of data.

`uint32_t sdaGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SDA signal. Set to 0 to disable.

`uint32_t sclGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SCL signal. Set to 0 to disable.

`uint32_t dataValidDelay_ns`
Width in nanoseconds of the data valid delay.

`uint32_t clockHoldTime_ns`
Width in nanoseconds of the clock hold time.

`struct _lpi2c_slave_transfer`
#include <fsl_lpi2c.h> LPI2C slave transfer structure.

Public Members

`lpi2c_slave_transfer_event_t event`
Reason the callback is being invoked.

`uint8_t receivedAddress`
Matching address send by master.

`uint8_t *data`
Transfer buffer

`size_t dataSize`
Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kLPI2C_SlaveCompletionEvent.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct *_lpi2c_slave_handle*

#include <fsl_lpi2c.h> LPI2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

lpi2c_slave_transfer_t transfer

LPI2C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

lpi2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

struct sclStall

Public Members

bool enableAck

Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted. Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDatasCLStall or enableAddressSCLStall.

bool enableTx

Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.

bool enableRx

Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.

bool enableAddress

Enables SCL clock stretching when the address valid flag is asserted.

2.45 LPIT: Low-Power Interrupt Timer

`void LPIT_Init(LPIT_Type *base, const lpit_config_t *config)`

Ungates the LPIT clock and configures the peripheral for a basic operation.

This function issues a software reset to reset all channels and registers except the Module Control register.

Note: This API should be called at the beginning of the application using the LPIT driver.

Parameters

- `base` – LPIT peripheral base address.
- `config` – Pointer to the user configuration structure.

`void LPIT_Deinit(LPIT_Type *base)`

Disables the module and gates the LPIT clock.

Parameters

- `base` – LPIT peripheral base address.

`void LPIT_GetDefaultConfig(lpit_config_t *config)`

Fills in the LPIT configuration structure with default settings.

The default values are:

```
config->enableRunInDebug = false;  
config->enableRunInDoze = false;
```

Parameters

- `config` – Pointer to the user configuration structure.

`status_t LPIT_SetupChannel(LPIT_Type *base, lpit_chnl_t channel, const lpit_chnl_params_t *chnlSetup)`

Sets up an LPIT channel based on the user's preference.

This function sets up the operation mode to one of the options available in the enumeration `lpit_timer_modes_t`. It sets the trigger source as either internal or external, trigger selection and the timers behaviour when a timeout occurs. It also chains the timer if a prior timer is requested by the user.

Parameters

- `base` – LPIT peripheral base address.
- `channel` – Channel that is being configured.
- `chnlSetup` – Configuration parameters.

`static inline void LPIT_EnableInterrupts(LPIT_Type *base, uint32_t mask)`

Enables the selected PIT interrupts.

Parameters

- `base` – LPIT peripheral base address.
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `lpit_interrupt_enable_t`

```
static inline void LPIT_DisableInterrupts(LPIT_Type *base, uint32_t mask)
```

Disables the selected PIT interrupts.

Parameters

- base – LPIT peripheral base address.
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `lpit_interrupt_enable_t`

```
static inline uint32_t LPIT_GetEnabledInterrupts(LPIT_Type *base)
```

Gets the enabled LPIT interrupts.

Parameters

- base – LPIT peripheral base address.

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `lpit_interrupt_enable_t`

```
static inline uint32_t LPIT_GetStatusFlags(LPIT_Type *base)
```

Gets the LPIT status flags.

Parameters

- base – LPIT peripheral base address.

Returns

The status flags. This is the logical OR of members of the enumeration `lpit_status_flags_t`

```
static inline void LPIT_ClearStatusFlags(LPIT_Type *base, uint32_t mask)
```

Clears the LPIT status flags.

Parameters

- base – LPIT peripheral base address.
- mask – The status flags to clear. This is a logical OR of members of the enumeration `lpit_status_flags_t`

```
static inline void LPIT_SetTimerPeriod(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

Timers begin counting down from the value set by this function until it reaches 0, at which point it generates an interrupt and loads this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

```
static inline void LPIT_SetTimerValue(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

In the Dual 16-bit Periodic Counter mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.

Note: Set TVAL register to 0 or 1 is invalid in compare mode.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

static inline uint32_t LPIT_GetCurrentTimerCount(LPIT_Type *base, *lpit_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to microseconds or milliseconds.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

Returns

Current timer counting value in ticks.

static inline void LPIT_StartTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Starts the timer counting.

After calling this function, timers load the period value and count down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

static inline void LPIT_StopTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Stops the timer counting.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

FSL_LPIT_DRIVER_VERSION

Version 2.1.3

enum *_lpit_chnl*

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

Values:

enumerator `kLPIT_Chnl_0`

LPIT channel number 0

enumerator kLPIT_Chnl_1
LPIT channel number 1

enumerator kLPIT_Chnl_2
LPIT channel number 2

enumerator kLPIT_Chnl_3
LPIT channel number 3

enum _lpit_timer_modes

Mode options available for the LPIT timer.

Values:

enumerator kLPIT_PeriodicCounter
Use the all 32-bits, counter loads and decrements to zero

enumerator kLPIT_DualPeriodicCounter
Counter loads, lower 16-bits decrement to zero, then upper 16-bits decrement

enumerator kLPIT_TriggerAccumulator
Counter loads on first trigger and decrements on each trigger

enumerator kLPIT_InputCapture
Counter loads with 0xFFFFFFFF, decrements to zero. It stores the inverse of the current value when a input trigger is detected

enum _lpit_trigger_select

Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

Values:

enumerator kLPIT_Trigger_TimerChn0
Channel 0 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn1
Channel 1 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn2
Channel 2 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn3
Channel 3 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn4
Channel 4 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn5
Channel 5 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn6
Channel 6 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn7
Channel 7 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn8
Channel 8 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn9
Channel 9 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn10

Channel 10 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn11

Channel 11 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn12

Channel 12 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn13

Channel 13 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn14

Channel 14 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn15

Channel 15 is selected as a trigger source

enum _lpit_trigger_source

Trigger source options available.

Values:

enumerator kLPIT_TriggerSource_External

Use external trigger input

enumerator kLPIT_TriggerSource_Internal

Use internal trigger

enum _lpit_interrupt_enable

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerInterruptEnable

Channel 0 Timer interrupt

enumerator kLPIT_Channel1TimerInterruptEnable

Channel 1 Timer interrupt

enumerator kLPIT_Channel2TimerInterruptEnable

Channel 2 Timer interrupt

enumerator kLPIT_Channel3TimerInterruptEnable

Channel 3 Timer interrupt

enum _lpit_status_flags

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerFlag

Channel 0 Timer interrupt flag

enumerator kLPIT_Channel1TimerFlag

Channel 1 Timer interrupt flag

enumerator kLPIT_Channel2TimerFlag

Channel 2 Timer interrupt flag

enumerator kLPIT_Channel3TimerFlag

Channel 3 Timer interrupt flag

typedef enum *_lpit_chnl* lpit_chnl_t

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

typedef enum *_lpit_timer_modes* lpit_timer_modes_t

Mode options available for the LPIT timer.

typedef enum *_lpit_trigger_select* lpit_trigger_select_t

Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

typedef enum *_lpit_trigger_source* lpit_trigger_source_t

Trigger source options available.

typedef enum *_lpit_interrupt_enable* lpit_interrupt_enable_t

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef enum *_lpit_status_flags* lpit_status_flags_t

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef struct *_lpit_chnl_params* lpit_chnl_params_t

Structure to configure the channel timer.

typedef struct *_lpit_config* lpit_config_t

LPIT configuration structure.

This structure holds the configuration settings for the LPIT peripheral. To initialize this structure to reasonable defaults, call the LPIT_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

static void LPIT_ResetStateDelay(void)

Short wait for LPIT state reset.

After clear or set LPIT_EN, there should be delay longer than 4 LPIT functional clock.

static inline void LPIT_Reset(LPIT_Type *base)

Performs a software reset on the LPIT module.

This resets all channels and registers except the Module Control Register.

Parameters

- base – LPIT peripheral base address.

LPIT_RESET_STATE_DELAY

Delay used in LPIT_Reset.

The macro value should be larger than $4 * \text{core clock} / \text{LPIT peripheral clock}$.

struct __lpit_chnl_params

#include <fsl_lpit.h> Structure to configure the channel timer.

Public Members

bool chainChannel

true: Timer chained to previous timer; false: Timer not chained

lpit_timer_modes_t timerMode

Timers mode of operation.

lpit_trigger_select_t triggerSelect

Trigger selection for the timer

lpit_trigger_source_t triggerSource

Decides if we use external or internal trigger.

bool enableReloadOnTrigger

true: Timer reloads when a trigger is detected; false: No effect

bool enableStopOnTimeout

true: Timer will stop after timeout; false: does not stop after timeout

bool enableStartOnTrigger

true: Timer starts when a trigger is detected; false: decrement immediately

struct __lpit_config

#include <fsl_lpit.h> LPIT configuration structure.

This structure holds the configuration settings for the LPIT peripheral. To initialize this structure to reasonable defaults, call the LPIT_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

bool enableRunInDoze

true: Timers run in doze mode; false: Timers stop in doze mode

2.46 LPSPI: Low Power Serial Peripheral Interface

2.47 LPSPI Peripheral driver

void LPSPI_MasterInit(LPSPI_Type *base, const lpspi_master_config_t *masterConfig, uint32_t srcClock_Hz)

Initializes the LPSPI master.

Parameters

- base – LPSPI peripheral address.
- masterConfig – Pointer to structure `lpspi_master_config_t`.
- srcClock_Hz – Module source input clock in Hertz

`void LPSPI_MasterGetDefaultConfig(lpspi_master_config_t *masterConfig)`

Sets the `lpspi_master_config_t` structure to default values.

This API initializes the configuration structure for `LPSPI_MasterInit()`. The initialized structure can remain unchanged in `LPSPI_MasterInit()`, or can be modified before calling the `LPSPI_MasterInit()`. Example:

```
lpspi_master_config_t masterConfig;
LPSPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – pointer to `lpspi_master_config_t` structure

`void LPSPI_SlaveInit(LPSPI_Type *base, const lpspi_slave_config_t *slaveConfig)`

LPSPi slave configuration.

Parameters

- base – LPSPi peripheral address.
- slaveConfig – Pointer to a structure `lpspi_slave_config_t`.

`void LPSPI_SlaveGetDefaultConfig(lpspi_slave_config_t *slaveConfig)`

Sets the `lpspi_slave_config_t` structure to default values.

This API initializes the configuration structure for `LPSPI_SlaveInit()`. The initialized structure can remain unchanged in `LPSPI_SlaveInit()` or can be modified before calling the `LPSPI_SlaveInit()`. Example:

```
lpspi_slave_config_t slaveConfig;
LPSPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – pointer to `lpspi_slave_config_t` structure.

`void LPSPI_Deinit(LPSPI_Type *base)`

De-initializes the LPSPi peripheral. Call this API to disable the LPSPi clock.

Parameters

- base – LPSPi peripheral address.

`void LPSPI_Reset(LPSPI_Type *base)`

Restores the LPSPi peripheral to reset state. Note that this function sets all registers to reset state. As a result, the LPSPi module can't work after calling this API.

Parameters

- base – LPSPi peripheral address.

`uint32_t LPSPI_GetInstance(LPSPI_Type *base)`

Get the LPSPi instance from peripheral base address.

Parameters

- base – LPSPi peripheral base address.

Returns

LPSPi instance.

```
static inline void LPSPI_Enable(LPSPI_Type *base, bool enable)
```

Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Parameters

- base – LPSPI peripheral address.
- enable – Pass true to enable module, false to disable module.

```
static inline uint32_t LPSPI_GetStatusFlags(LPSPI_Type *base)
```

Gets the LPSPI status flag state.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI status(in SR register).

```
static inline uint8_t LPSPI_GetTxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Tx FIFO size.

```
static inline uint8_t LPSPI_GetRxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Rx FIFO size.

```
static inline uint32_t LPSPI_GetTxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the transmit FIFO.

```
static inline uint32_t LPSPI_GetRxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the receive FIFO.

```
static inline void LPSPI_ClearStatusFlags(LPSPI_Type *base, uint32_t statusFlags)
```

Clears the LPSPI status flag.

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the `_lpspi_flags`. Example usage:

```
LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag|kLPSPI_RxDataReadyFlag);
```

Parameters

- base – LPSPI peripheral address.
- statusFlags – The status flag used from type `_lpspi_flags`.

static inline uint32_t LPSPI_GetTcr(LPSPI_Type *base)

static inline void LPSPI_EnableInterrupts(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI interrupts.

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_DisableInterrupts(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI interrupts.

```
LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_EnableDMA(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

static inline void LPSPI_DisableDMA(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
SPI_DisableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

static inline uint32_t LPSPI_GetTxRegisterAddress(LPSPI_Type *base)

Gets the LPSPI Transmit Data Register address for a DMA operation.

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Transmit Data Register address.

```
static inline uint32_t LPSPI_GetRxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Receive Data Register address for a DMA operation.

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Receive Data Register address.

```
bool LPSPI_CheckTransferArgument(LPSPI_Type *base, lpspi_transfer_t *transfer, bool isEdma)
```

Check the argument for transfer .

Parameters

- base – LPSPI peripheral address.
- transfer – the transfer struct to be used.
- isEdma – True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns

Return true for right and false for wrong.

```
static inline void LPSPI_SetMasterSlaveMode(LPSPI_Type *base, lpspi_master_slave_mode_t mode)
```

Configures the LPSPI for either master or slave.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

- base – LPSPI peripheral address.
- mode – Mode setting (master or slave) of type lpspi_master_slave_mode_t.

```
static inline void LPSPI_SelectTransferPCS(LPSPI_Type *base, lpspi_which_pcs_t select)
```

Configures the peripheral chip select used for the transfer.

Parameters

- base – LPSPI peripheral address.
- select – LPSPI Peripheral Chip Select (PCS) configuration.

```
static inline void LPSPI_SetPCSContinuous(LPSPI_Type *base, bool IsContinuous)
```

Set the PCS signal to continuous or uncontinuous mode.

Note: In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

- base – LPSPI peripheral address.
- IsContinuous – True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

static inline bool LPSPI_IsMaster(LPSPI_Type *base)

Returns whether the LPSPI module is in master mode.

Parameters

- base – LPSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

static inline void LPSPI_FlushFifo(LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)

Flushes the LPSPI FIFOs.

Parameters

- base – LPSPI peripheral address.
- flushTxFifo – Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
- flushRxFifo – Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

static inline void LPSPI_SetFifoWatermarks(LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)

Sets the transmit and receive FIFO watermark values.

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

- base – LPSPI peripheral address.
- txWater – The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
- rxWater – The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPSPI_SetAllPcsPolarity(LPSPI_Type *base, uint32_t mask)

Configures all LPSPI peripheral chip select polarities simultaneously.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow | kLPSPI_Pcs1ActiveLow);
```

Parameters

- base – LPSPI peripheral address.
- mask – The PCS polarity mask; Use the enum `_lpspi_pcs_polarity`.

static inline void LPSPI_SetFrameSize(LPSPI_Type *base, uint32_t frameSize)

Configures the frame size.

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame

size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPI in slave mode, although the command register does not update until after the LPSPI is enabled. After it is enabled, the transmit command register should only be changed if the LPSPI is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

- base – LPSPI peripheral address.
- frameSize – The frame size in number of bits.

```
uint32_t LPSPI_MasterSetBaudRate(LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t  
                                srcClock_Hz, uint32_t *tcrPrescaleValue)
```

Sets the LPSPI baud rate in bits per second.

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale tcrPrescaleValue parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- baudRate_Bps – The desired baud rate in bits per second.
- srcClock_Hz – Module source input clock in Hertz.
- tcrPrescaleValue – The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a “0” if the LPSPI is not configured for master mode or if the LPSPI module is not disabled.

```
void LPSPI_MasterSetDelayScaler(LPSPI_Type *base, uint32_t scaler, lpspi_delay_type_t  
                                whichDelay)
```

Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type *lpspi_delay_type_t*.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.

- `scaler` – The 8-bit delay value 0x00 to 0xFF (255).
- `whichDelay` – The desired delay to configure, must be of type `lpspi_delay_type_t`.

```
uint32_t LPSPI_MasterSetDelayTimes(LPSPI_Type *base, uint32_t delayTimeInNanoSec,
                                   lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)
```

Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPI module must be configured for master mode before configuring this. And note that the `delayTime = LPSPI_clockSource / (PRESCALE * Delay_scaler)`.

Parameters

- `base` – LPSPI peripheral address.
- `delayTimeInNanoSec` – The desired delay value in nano-seconds.
- `whichDelay` – The desired delay to configuration, which must be of type `lpspi_delay_type_t`.
- `srcClock_Hz` – Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

```
static inline void LPSPI_WriteData(LPSPI_Type *base, uint32_t data)
```

Writes data into the transmit data buffer.

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

- `base` – LPSPI peripheral address.
- `data` – The data word to be sent.

```
static inline uint32_t LPSPI_ReadData(LPSPI_Type *base)
```

Reads data from the data buffer.

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

- `base` – LPSPI peripheral address.

Returns

The data read from the data buffer.

```
void LPSPI_SetDummyData(LPSPI_Type *base, uint8_t dummyData)
```

Set up the dummy data.

Parameters

- *base* – LPSPI peripheral address.
- *dummyData* – Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

```
void LPSPI_MasterTransferCreateHandle(LPSPI_Type *base, lpspi_master_handle_t *handle,  
                                     lpspi_master_transfer_callback_t callback, void  
                                     *userData)
```

Initializes the LPSPI master handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – LPSPI handle pointer to *lpspi_master_handle_t*.
- *callback* – DSPI callback.
- *userData* – callback function parameter.

```
status_t LPSPI_MasterTransferBlocking(LPSPI_Type *base, lpspi_transfer_t *transfer)
```

LPSPI master transfer data using a polling method.

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *transfer* – pointer to *lpspi_transfer_t* structure.

Returns

status of status_t.

```
status_t LPSPI_MasterTransferNonBlocking(LPSPI_Type *base, lpspi_master_handle_t *handle,  
                                         lpspi_transfer_t *transfer)
```

LPSPI master transfer data using an interrupt method.

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – pointer to *lpspi_master_handle_t* structure which stores the transfer state.

- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_MasterTransferGetCount(LPSPI_Type *base, *lpspi_master_handle_t* *handle, size_t *count)

Gets the master transfer remaining bytes.

This function gets the master transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

void LPSPI_MasterTransferAbort(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI master abort transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

void LPSPI_MasterTransferHandleIRQ(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI Master IRQ handler function.

This function processes the LPSPI transmit and receive IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

void LPSPI_SlaveTransferCreateHandle(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_slave_transfer_callback_t* callback, void *userData)

Initializes the LPSPI slave handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to `lpspi_slave_handle_t`.
- callback – DSPI callback.
- userData – callback function parameter.

`status_t` LPSPI_SlaveTransferNonBlocking(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfer data using an interrupt method.

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_SlaveTransferGetCount(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, size_t *count)

Gets the slave transfer remaining bytes.

This function gets the slave transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

void LPSPI_SlaveTransferAbort(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI slave aborts a transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

void LPSPI_SlaveTransferHandleIRQ(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI Slave IRQ handler function.

This function processes the LPSPI transmit and receives an IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

bool LPSPI_WaitTxFifoEmpty(LPSPI_Type *base)

Wait for tx FIFO to be empty.

This function wait the tx fifo empty

Parameters

- base – LPSPI peripheral address.

Returns

true for the tx FIFO is ready, false is not.

void LPSPI_DriverIRQHandler(uint32_t instance)

LPSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for LPSPI.

Parameters

- instance – LPSPI instance.

FSL_LPSPI_DRIVER_VERSION

LPSPI driver version.

Status for the LPSPI driver.

Values:

enumerator kStatus_LPSPI_Busy

LPSPI transfer is busy.

enumerator kStatus_LPSPI_Error

LPSPI driver error.

enumerator kStatus_LPSPI_Idle

LPSPI is idle.

enumerator kStatus_LPSPI_OutOfRange

LPSPI transfer out Of range.

enumerator kStatus_LPSPI_Timeout

LPSPI timeout polling status flags.

enum _lpspi_flags

LPSPI status flags in SPIx_SR register.

Values:

enumerator kLPSPI_TxDataRequestFlag

Transmit data flag

enumerator kLPSPI_RxDataReadyFlag

Receive data flag

enumerator kLPSPI_WordCompleteFlag

Word Complete flag

enumerator kLPSPI_FrameCompleteFlag

Frame Complete flag

enumerator kLPSPI_TransferCompleteFlag

Transfer Complete flag

enumerator kLPSPI_TransmitErrorFlag

Transmit Error flag (FIFO underrun)

enumerator kLPSPI_ReceiveErrorFlag

Receive Error flag (FIFO overrun)

enumerator kLPSPI_DataMatchFlag

Data Match flag

enumerator kLPSPI_ModuleBusyFlag

Module Busy flag

enumerator kLPSPI_AllStatusFlag

Used for clearing all w1c status flags

enum _lpspi_interrupt_enable

LPSPI interrupt source.

Values:

enumerator kLPSPI_TxInterruptEnable

Transmit data interrupt enable

enumerator kLPSPI_RxInterruptEnable

Receive data interrupt enable

enumerator kLPSPI_WordCompleteInterruptEnable

Word complete interrupt enable

enumerator kLPSPI_FrameCompleteInterruptEnable

Frame complete interrupt enable

enumerator kLPSPI_TransferCompleteInterruptEnable

Transfer complete interrupt enable

enumerator kLPSPI_TransmitErrorInterruptEnable

Transmit error interrupt enable(FIFO underrun)

enumerator kLPSPI_ReceiveErrorInterruptEnable

Receive Error interrupt enable (FIFO overrun)

enumerator kLPSPI_DataMatchInterruptEnable

Data Match interrupt enable

enumerator kLPSPI_AllInterruptEnable

All above interrupts enable.

enum _lpspi_dma_enable

LPSPI DMA source.

Values:

enumerator kLPSPI_TxDmaEnable

Transmit data DMA enable

enumerator kLPSPI_RxDmaEnable

Receive data DMA enable

enum _lpspi_master_slave_mode

LPSPI master or slave mode configuration.

Values:

enumerator kLPSPI_Master

LPSPI peripheral operates in master mode.

enumerator kLPSPI_Slave

LPSPI peripheral operates in slave mode.

enum _lpspi_which_pcs_config

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

Values:

enumerator kLPSPI_Pcs0

PCS[0]

enumerator kLPSPI_Pcs1
PCS[1]

enumerator kLPSPI_Pcs2
PCS[2]

enumerator kLPSPI_Pcs3
PCS[3]

enum __lpspi_pcs_polarity_config
LPSPI Peripheral Chip Select (PCS) Polarity configuration.
Values:

enumerator kLPSPI_PcsActiveHigh
PCS Active High (idles low)

enumerator kLPSPI_PcsActiveLow
PCS Active Low (idles high)

enum __lpspi_pcs_polarity
LPSPI Peripheral Chip Select (PCS) Polarity.
Values:

enumerator kLPSPI_Pcs0ActiveLow
Pcs0 Active Low (idles high).

enumerator kLPSPI_Pcs1ActiveLow
Pcs1 Active Low (idles high).

enumerator kLPSPI_Pcs2ActiveLow
Pcs2 Active Low (idles high).

enumerator kLPSPI_Pcs3ActiveLow
Pcs3 Active Low (idles high).

enumerator kLPSPI_PcsAllActiveLow
Pcs0 to Pcs5 Active Low (idles high).

enum __lpspi_clock_polarity
LPSPI clock polarity configuration.
Values:

enumerator kLPSPI_ClockPolarityActiveHigh
CPOL=0. Active-high LPSPI clock (idles low)

enumerator kLPSPI_ClockPolarityActiveLow
CPOL=1. Active-low LPSPI clock (idles high)

enum __lpspi_clock_phase
LPSPI clock phase configuration.
Values:

enumerator kLPSPI_ClockPhaseFirstEdge
CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

enumerator kLPSPI_ClockPhaseSecondEdge
CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

enum `_lpspi_shift_direction`

LPSPi data shifter direction options.

Values:

enumerator `kLPSPi_MsbFirst`

Data transfers start with most significant bit.

enumerator `kLPSPi_LsbFirst`

Data transfers start with least significant bit.

enum `_lpspi_host_request_select`

LPSPi Host Request select configuration.

Values:

enumerator `kLPSPi_HostReqExtPin`

Host Request is an ext pin.

enumerator `kLPSPi_HostReqInternalTrigger`

Host Request is an internal trigger.

enum `_lpspi_match_config`

LPSPi Match configuration options.

Values:

enumerator `kLPSPi_MatchDisabled`

LPSPi Match Disabled.

enumerator `kLPSPi_1stWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSPi_AnyWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSPi_1stWordEqualsM0and2ndWordEqualsM1`

LPSPi Match Enabled.

enumerator `kLPSPi_AnyWordEqualsM0andNxtWordEqualsM1`

LPSPi Match Enabled.

enumerator `kLPSPi_1stWordAndM1EqualsM0andM1`

LPSPi Match Enabled.

enumerator `kLPSPi_AnyWordAndM1EqualsM0andM1`

LPSPi Match Enabled.

enum `_lpspi_pin_config`

LPSPi pin (SDO and SDI) configuration.

Values:

enumerator `kLPSPi_SdiInSdoOut`

LPSPi SDI input, SDO output.

enumerator `kLPSPi_SdiInSdiOut`

LPSPi SDI input, SDI output.

enumerator `kLPSPi_SdoInSdoOut`

LPSPi SDO input, SDO output.

enumerator `kLPSPi_SdoInSdiOut`

LPSPi SDO input, SDI output.

enum `_lpspi_data_out_config`

LPSPi data output configuration.

Values:

enumerator `kLpspiDataOutRetained`

Data out retains last value when chip select is de-asserted

enumerator `kLpspiDataOutTristate`

Data out is tristated when chip select is de-asserted

enum `_lpspi_transfer_width`

LPSPi transfer width configuration.

Values:

enumerator `kLPSPi_SingleBitXfer`

1-bit shift at a time, data out on SDO, in on SDI (normal mode)

enumerator `kLPSPi_TwoBitXfer`

2-bits shift out on SDO/SDI and in on SDO/SDI

enumerator `kLPSPi_FourBitXfer`

4-bits shift out on SDO/SDI/PCS[3:2] and in on SDO/SDI/PCS[3:2]

enum `_lpspi_delay_type`

LPSPi delay type selection.

Values:

enumerator `kLPSPi_PcsToSck`

PCS-to-SCK delay.

enumerator `kLPSPi_LastSckToPcs`

Last SCK edge to PCS delay.

enumerator `kLPSPi_BetweenTransfer`

Delay between transfers.

enum `_lpspi_transfer_config_flag_for_master`

Use this enumeration for LPSPi master transfer configFlags.

Values:

enumerator `kLPSPi_MasterPcs0`

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS0 signal

enumerator `kLPSPi_MasterPcs1`

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS1 signal

enumerator `kLPSPi_MasterPcs2`

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS2 signal

enumerator `kLPSPi_MasterPcs3`

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS3 signal

enumerator `kLPSPi_MasterPcsContinuous`

Is PCS signal continuous

enumerator `kLPSPi_MasterByteSwap`

Is master swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set `lpspi_shift_direction_t` to MSB).

- i. If you set `bitPerFrame = 8` , no matter the `kLPSPi_MasterByteSwap` you flag is used or not, the waveform is 1 2 3 4 5 6 7 8.

- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.

enum _lpspi_transfer_config_flag_for_slave

Use this enumeration for LPSPi slave transfer configFlags.

Values:

enumerator kLPSPi_SlavePcs0

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS0 signal

enumerator kLPSPi_SlavePcs1

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS1 signal

enumerator kLPSPi_SlavePcs2

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS2 signal

enumerator kLPSPi_SlavePcs3

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS3 signal

enumerator kLPSPi_SlaveByteSwap

Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

- i. If you set bitPerFrame = 8 , no matter the kLPSPi_SlaveByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.

enum _lpspi_transfer_state

LPSPi transfer state, which is used for LPSPi transactional API state machine.

Values:

enumerator kLPSPi_Idle

Nothing in the transmitter/receiver.

enumerator kLPSPi_Busy

Transfer queue is not finished.

enumerator kLPSPi_Error

Transfer error.

typedef enum _lpspi_master_slave_mode lpspi_master_slave_mode_t

LPSPi master or slave mode configuration.

typedef enum _lpspi_which_pcs_config lpspi_which_pcs_t

LPSPi Peripheral Chip Select (PCS) configuration (which PCS to configure).

typedef enum _lpspi_pcs_polarity_config lpspi_pcs_polarity_config_t

LPSPi Peripheral Chip Select (PCS) Polarity configuration.


```
typedef enum _lpspi_clock_polarity lpspi_clock_polarity_t
    LPSPI clock polarity configuration.

typedef enum _lpspi_clock_phase lpspi_clock_phase_t
    LPSPI clock phase configuration.

typedef enum _lpspi_shift_direction lpspi_shift_direction_t
    LPSPI data shifter direction options.

typedef enum _lpspi_host_request_select lpspi_host_request_select_t
    LPSPI Host Request select configuration.

typedef enum _lpspi_match_config lpspi_match_config_t
    LPSPI Match configuration options.

typedef enum _lpspi_pin_config lpspi_pin_config_t
    LPSPI pin (SDO and SDI) configuration.

typedef enum _lpspi_data_out_config lpspi_data_out_config_t
    LPSPI data output configuration.

typedef enum _lpspi_transfer_width lpspi_transfer_width_t
    LPSPI transfer width configuration.

typedef enum _lpspi_delay_type lpspi_delay_type_t
    LPSPI delay type selection.

typedef struct _lpspi_master_config lpspi_master_config_t
    LPSPI master configuration structure.

typedef struct _lpspi_slave_config lpspi_slave_config_t
    LPSPI slave configuration structure.

typedef struct _lpspi_master_handle lpspi_master_handle_t
    Forward declaration of the _lpspi_master_handle typedefs.

typedef struct _lpspi_slave_handle lpspi_slave_handle_t
    Forward declaration of the _lpspi_slave_handle typedefs.

typedef void (*lpspi_master_transfer_callback_t)(LPSPi_Type *base, lpspi_master_handle_t
*handle, status_t status, void *userData)
    Master completion callback function pointer type.

    Param base
        LPSPI peripheral address.

    Param handle
        Pointer to the handle for the LPSPI master.

    Param status
        Success or error code describing whether the transfer is completed.

    Param userData
        Arbitrary pointer-dataSized value passed from the application.

typedef void (*lpspi_slave_transfer_callback_t)(LPSPi_Type *base, lpspi_slave_handle_t *handle,
status_t status, void *userData)
    Slave completion callback function pointer type.

    Param base
        LPSPI peripheral address.

    Param handle
        Pointer to the handle for the LPSPI slave.
```

Param status

Success or error code describing whether the transfer is completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

`typedef struct _lpspi_transfer lpspi_transfer_t`

LPSPI master/slave transfer structure.

`volatile uint8_t g_lpspiDummyData[]`

Global variable for dummy data value setting.

`LPSPI_DUMMY_DATA`

LPSPI dummy data if no Tx data.

Dummy data used for tx if there is not txData.

`SPI_RETRY_TIMES`

Retry times for waiting flag.

`LPSPI_MASTER_PCS_SHIFT`

LPSPI master PCS shift macro , internal used.

`LPSPI_MASTER_PCS_MASK`

LPSPI master PCS shift macro , internal used.

`LPSPI_SLAVE_PCS_SHIFT`

LPSPI slave PCS shift macro , internal used.

`LPSPI_SLAVE_PCS_MASK`

LPSPI slave PCS shift macro , internal used.

`struct _lpspi_master_config`

`#include <fsl_lpspi.h>` LPSPI master configuration structure.

Public Members

`uint32_t baudRate`

Baud Rate for LPSPI.

`uint32_t bitsPerFrame`

Bits per frame, minimum 8, maximum 4096.

`lpspi_clock_polarity_t cpol`

Clock polarity.

`lpspi_clock_phase_t cpha`

Clock phase.

`lpspi_shift_direction_t direction`

MSB or LSB data shift direction.

`uint32_t pcsToSckDelayInNanoSec`

PCS to SCK delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

`uint32_t lastSckToPcsDelayInNanoSec`

Last SCK to PCS delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

uint32_t betweenTransferDelayInNanoSec

After the SCK delay time with nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

lpspi_which_pcs_t whichPcs

Desired Peripheral Chip Select (PCS).

lpspi_pcs_polarity_config_t pcsActiveHighOrLow

Desired PCS active high or low

lpspi_pin_config_t pinCfg

Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPi_PCS is negated).

bool enableInputDelay

Enable master to sample the input data on a delayed SCK. This can help improve slave setup time. Refer to device data sheet for specific time length.

struct __lpspi_slave_config

#include <fsl_lpspi.h> LPSPi slave configuration structure.

Public Members

uint32_t bitsPerFrame

Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol

Clock polarity.

lpspi_clock_phase_t cpha

Clock phase.

lpspi_shift_direction_t direction

MSB or LSB data shift direction.

lpspi_which_pcs_t whichPcs

Desired Peripheral Chip Select (pcs)

lpspi_pcs_polarity_config_t pcsActiveHighOrLow

Desired PCS active high or low

lpspi_pin_config_t pinCfg

Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPi_PCS is negated).

struct __lpspi_transfer

#include <fsl_lpspi.h> LPSPi master/slave transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

volatile size_t dataSize

Transfer bytes.

uint32_t configFlags

Transfer transfer configuration flags. Set from `_lpspi_transfer_config_flag_for_master` if the transfer is used for master or `_lpspi_transfer_config_flag_for_slave` enumeration if the transfer is used for slave.

struct `_lpspi_master_handle`

#include <fsl_lpspi.h> LPSPi master transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous

Is PCS continuous in transfer.

volatile bool writeTcrInIsr

A flag that whether should write TCR in ISR.

volatile bool isByteSwap

A flag that whether should byte swap.

volatile bool isTxMask

A flag that whether TCR[TXMSK] is set.

volatile uint16_t bytesPerFrame

Number of bytes in each frame

volatile uint16_t frameSize

Backup of TCR[FRAMESZ]

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

const uint8_t *volatile txData

Send buffer.

uint8_t *volatile rxData

Receive buffer.

volatile size_t txRemainingByteCount

Number of bytes remaining to send.

volatile size_t rxRemainingByteCount

Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes

Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes

Read RDR register remaining times.

uint32_t totalByteCount

Number of transfer bytes

uint32_t txBuffIfNull

Used if the txData is NULL.

volatile uint8_t state

LPSPi transfer state , _lpspi_transfer_state.

lpspi_master_transfer_callback_t callback

Completion callback.

void *userData

Callback user data.

struct _lpspi_slave_handle

#include <fsl_lpspi.h> LPSPi slave transfer handle structure used for transactional API.

Public Members

volatile bool isByteSwap

A flag that whether should byte swap.

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

const uint8_t *volatile txData

Send buffer.

uint8_t *volatile rxData

Receive buffer.

volatile size_t txRemainingByteCount

Number of bytes remaining to send.

volatile size_t rxRemainingByteCount

Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes

Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes

Read RDR register remaining times.

uint32_t totalByteCount

Number of transfer bytes

volatile uint8_t state

LPSPi transfer state , _lpspi_transfer_state.

volatile uint32_t errorCount

Error count for slave transfer.

lpspi_slave_transfer_callback_t callback

Completion callback.

void *userData

Callback user data.

2.48 LPSPI eDMA Driver

FSL_LPSPI_EDMA_DRIVER_VERSION

LPSPI EDMA driver version.

DMA_MAX_TRANSFER_COUNT

DMA max transfer size.

typedef struct *lpspi_master_edma_handle* lpspi_master_edma_handle_t

Forward declaration of the *lpspi_master_edma_handle* typedefs.

typedef struct *lpspi_slave_edma_handle* lpspi_slave_edma_handle_t

Forward declaration of the *lpspi_slave_edma_handle* typedefs.

typedef void (*lpspi_master_edma_transfer_callback_t)(LPSPI_Type *base,
lpspi_master_edma_handle_t *handle, *status_t* status, void *userData)

Completion callback function pointer type.

Param base

LPSPI peripheral base address.

Param handle

Pointer to the handle for the LPSPI master.

Param status

Success or error code describing whether the transfer completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

typedef void (*lpspi_slave_edma_transfer_callback_t)(LPSPI_Type *base,
lpspi_slave_edma_handle_t *handle, *status_t* status, void *userData)

Completion callback function pointer type.

Param base

LPSPI peripheral base address.

Param handle

Pointer to the handle for the LPSPI slave.

Param status

Success or error code describing whether the transfer completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

void LPSPI_MasterTransferCreateHandleEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t*
*handle, *lpspi_master_edma_transfer_callback_t*
callback, void *userData, *edma_handle_t*
*edmaRxRegToRxDataHandle, *edma_handle_t*
*edmaTxDataToTxRegHandle)

Initializes the LPSPI master eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that the LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx are the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for `edmaRxRegToRxDataHandle` and Tx DMAMUX source for `edmaTxDataToTxRegHandle`. (2) For a shared DMA request source, enable and set the Rx/Tx DMAMUX source for `edmaRxRegToRxDataHandle`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – LPSPI handle pointer to `lpspi_master_edma_handle_t`.
- `callback` – LPSPI callback.
- `userData` – callback function parameter.
- `edmaRxRegToRxDataHandle` – `edmaRxRegToRxDataHandle` pointer to `edma_handle_t`.
- `edmaTxDataToTxRegHandle` – `edmaTxDataToTxRegHandle` pointer to `edma_handle_t`.

status_t LPSPI_MasterTransferEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of `bytesPerFrame` if `bytesPerFrame` is less than or equal to 4. For `bytesPerFrame` greater than 4: The transfer data size should be equal to `bytesPerFrame` if the `bytesPerFrame` is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of `bytesPerFrame`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- `transfer` – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

status_t LPSPI_MasterTransferPrepareEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, `uint32_t` configFlags)

LPSPI master config transfer parameter while using eDMA.

This function is preparing to transfer data using eDMA, work with `LPSPI_MasterTransferEDMALite`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- `configFlags` – transfer configuration flags. `_lpspi_transfer_config_flag_for_master`.

Return values

- `kStatus_Success` – Execution successfully.
- `kStatus_LPSPI_Busy` – The LPSPI device is busy.

Returns

Indicates whether LPSPI master transfer was successful or not.

status_t LPSPI_MasterTransferEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA without configs.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: This API is only for transfer through DMA without configuration. Before calling this API, you must call LPSPI_MasterTransferPrepareEDMALite to configure it once. The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to *lpspi_master_edma_handle_t* structure which stores the transfer state.
- transfer – pointer to *lpspi_transfer_t* structure, config field is not used.

Return values

- kStatus_Success – Execution successfully.
- kStatus_LPSPI_Busy – The LPSPI device is busy.
- kStatus_InvalidArgument – The transfer structure is invalid.

Returns

Indicates whether LPSPI master transfer was successful or not.

void LPSPI_MasterTransferAbortEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle)

LPSPI master aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to *lpspi_master_edma_handle_t* structure which stores the transfer state.

status_t LPSPI_MasterTransferGetCountEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *size_t* *count)

Gets the master eDMA transfer remaining bytes.

This function gets the master eDMA transfer remaining bytes.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to *lpspi_master_edma_handle_t* structure which stores the transfer state.
- count – Number of bytes transferred so far by the EDMA transaction.

Returns

status of *status_t*.


```
void LPSPI_SlaveTransferCreateHandleEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t
                                         *handle, lpspi_slave_edma_transfer_callback_t
                                         callback, void *userData, edma_handle_t
                                         *edmaRxRegToRxDataHandle, edma_handle_t
                                         *edmaTxDataToTxRegHandle)
```

Initializes the LPSPI slave eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx as the same source) DMA request source.

(1) For a separated DMA request source, enable and set the Rx DMAMUX source for `edmaRxRegToRxDataHandle` and Tx DMAMUX source for `edmaTxDataToTxRegHandle`. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for `edmaRxRegToRxDataHandle`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – LPSPI handle pointer to `lpspi_slave_edma_handle_t`.
- `callback` – LPSPI callback.
- `userData` – callback function parameter.
- `edmaRxRegToRxDataHandle` – `edmaRxRegToRxDataHandle` pointer to `edma_handle_t`.
- `edmaTxDataToTxRegHandle` – `edmaTxDataToTxRegHandle` pointer to `edma_handle_t`.

```
status_t LPSPI_SlaveTransferEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle,
                                  lpspi_transfer_t *transfer)
```

LPSPI slave transfers data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which return right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of `bytesPerFrame` if `bytesPerFrame` is less than or equal to 4. For `bytesPerFrame` greater than 4: The transfer data size should be equal to `bytesPerFrame` if the `bytesPerFrame` is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of `bytesPerFrame`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- `transfer` – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

```
void LPSPI_SlaveTransferAbortEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle)
```

LPSPI slave aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- `base` – LPSPI peripheral base address.

- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.

`status_t` LPSPI_SlaveTransferGetCountEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, `size_t` *count)

Gets the slave eDMA transfer remaining bytes.

This function gets the slave eDMA transfer remaining bytes.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the eDMA transaction.

Returns

status of `status_t`.

`struct _lpspi_master_edma_handle`

#include <fsl_lpspi_edma.h> LPSPI master eDMA transfer handle structure used for transactional API.

Public Members

`volatile bool` `isPcsContinuous`

Is PCS continuous in transfer.

`volatile bool` `isByteSwap`

A flag that whether should byte swap.

`volatile uint8_t` `fifoSize`

FIFO dataSize.

`volatile uint8_t` `rxWatermark`

Rx watermark.

`volatile uint8_t` `bytesEachWrite`

Bytes for each write TDR.

`volatile uint8_t` `bytesEachRead`

Bytes for each read RDR.

`volatile uint8_t` `bytesLastRead`

Bytes for last read RDR.

`volatile bool` `isThereExtraRxBytes`

Is there extra RX byte.

`const uint8_t *volatile` `txData`

Send buffer.

`uint8_t *volatile` `rxData`

Receive buffer.

`volatile size_t` `txRemainingByteCount`

Number of bytes remaining to send.

`volatile size_t` `rxRemainingByteCount`

Number of bytes remaining to receive.

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`edma_tcd_t *lastTimeTCD`
Pointer to the lastTime TCD

`bool isMultiDMATransmit`
Is there multi DMA transmit

`volatile uint8_t dmaTransmitTime`
DMA Transfer times.

`uint32_t lastTimeDataBytes`
DMA transmit last Time data Bytes

`uint32_t dataBytesEveryTime`
Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

`edma_transfer_config_t transferConfigRx`
Config of DMA rx channel.

`edma_transfer_config_t transferConfigTx`
Config of DMA tx channel.

`uint32_t txBuffIfNull`
Used if there is not txData for DMA purpose.

`uint32_t rxBuffIfNull`
Used if there is not rxData for DMA purpose.

`uint32_t transmitCommand`
Used to write TCR for DMA purpose.

`volatile uint8_t state`
LPSPI transfer state , `_lpspi_transfer_state`.

`uint8_t nbytes`
eDMA minor byte transfer count initially configured.

`lpspi_master_edma_transfer_callback_t callback`
Completion callback.

`void *userData`
Callback user data.

`edma_handle_t *edmaRxRegToRxDataHandle`
edma_handle_t handle point used for RxReg to RxData buff

`edma_handle_t *edmaTxDataToTxRegHandle`
edma_handle_t handle point used for TxData to TxReg buff

`edma_tcd_t lpspiSoftwareTCD[3]`
SoftwareTCD, internal used

`struct _lpspi_slave_edma_handle`
`#include <fsl_lpspi_edma.h>` LPSPI slave eDMA transfer handle structure used for transactional API.

Public Members

volatile bool isByteSwap

A flag that whether should byte swap.

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

volatile uint8_t bytesLastRead

Bytes for last read RDR.

volatile bool isThereExtraRxBytes

Is there extra RX byte.

uint8_t nbytes

eDMA minor byte transfer count initially configured.

const uint8_t *volatile txData

Send buffer.

uint8_t *volatile rxData

Receive buffer.

volatile size_t txRemainingByteCount

Number of bytes remaining to send.

volatile size_t rxRemainingByteCount

Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes

Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes

Read RDR register remaining times.

uint32_t totalByteCount

Number of transfer bytes

uint32_t txBuffIfNull

Used if there is not txData for DMA purpose.

uint32_t rxBuffIfNull

Used if there is not rxData for DMA purpose.

volatile uint8_t state

LPSPi transfer state.

uint32_t errorCount

Error count for slave transfer.

lpspi_slave_edma_transfer_callback_t callback

Completion callback.

`void *userData`

Callback user data.

`edma_handle_t *edmaRxRegToRxDataHandle`

`edma_handle_t` handle point used for RxReg to RxData buff

`edma_handle_t *edmaTxDataToTxRegHandle`

`edma_handle_t` handle point used for TxData to TxReg

`edma_tcd_t` `lpspiSoftwareTCD[2]`

SoftwareTCD, internal used

2.49 LPTMR: Low-Power Timer

`void LPTMR_Init(LPTMR_Type *base, const lptmr_config_t *config)`

Ungates the LPTMR clock and configures the peripheral for a basic operation.

Note: This API should be called at the beginning of the application using the LPTMR driver.

Parameters

- `base` – LPTMR peripheral base address
- `config` – A pointer to the LPTMR configuration structure.

`void LPTMR_Deinit(LPTMR_Type *base)`

Gates the LPTMR clock.

Parameters

- `base` – LPTMR peripheral base address

`void LPTMR_GetDefaultConfig(lptmr_config_t *config)`

Fills in the LPTMR configuration structure with default settings.

The default values are as follows.

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

- `config` – A pointer to the LPTMR configuration structure.

`static inline void LPTMR_EnableInterrupts(LPTMR_Type *base, uint32_t mask)`

Enables the selected LPTMR interrupts.

Parameters

- `base` – LPTMR peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `lptmr_interrupt_enable_t`

static inline void LPTMR_DisableInterrupts(LPTMR_Type *base, uint32_t mask)

Disables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `lptmr_interrupt_enable_t`.

static inline uint32_t LPTMR_GetEnabledInterrupts(LPTMR_Type *base)

Gets the enabled LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `lptmr_interrupt_enable_t`

static inline uint32_t LPTMR_GetStatusFlags(LPTMR_Type *base)

Gets the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `lptmr_status_flags_t`

static inline void LPTMR_ClearStatusFlags(LPTMR_Type *base, uint32_t mask)

Clears the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `lptmr_status_flags_t`.

static inline void LPTMR_SetTimerPeriod(LPTMR_Type *base, uint32_t ticks)

Sets the timer period in units of count.

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note:

- a. The TCF flag is set with the CNR equals the count provided here and then increments.
 - b. Call the utility macros provided in the `fsl_common.h` to convert to ticks.
-

Parameters

- base – LPTMR peripheral base address
- ticks – A timer period in units of ticks

static inline uint32_t LPTMR_GetCurrentTimerCount(LPTMR_Type *base)

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – LPTMR peripheral base address

Returns

The current counter value in ticks

static inline void LPTMR_StartTimer(LPTMR_Type *base)

Starts the timer.

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

- base – LPTMR peripheral base address

static inline void LPTMR_StopTimer(LPTMR_Type *base)

Stops the timer.

This function stops the timer and resets the timer's counter register.

Parameters

- base – LPTMR peripheral base address

FSL_LPTMR_DRIVER_VERSION

Driver Version

enum _lptmr_pin_select

LPTMR pin selection used in pulse counter mode.

Values:

enumerator kLPTMR_PinSelectInput_0

Pulse counter input 0 is selected

enumerator kLPTMR_PinSelectInput_1

Pulse counter input 1 is selected

enumerator kLPTMR_PinSelectInput_2

Pulse counter input 2 is selected

enumerator kLPTMR_PinSelectInput_3

Pulse counter input 3 is selected

enum _lptmr_pin_polarity

LPTMR pin polarity used in pulse counter mode.

Values:

enumerator kLPTMR_PinPolarityActiveHigh

Pulse Counter input source is active-high

enumerator kLPTMR_PinPolarityActiveLow

Pulse Counter input source is active-low

enum _lptmr_timer_mode

LPTMR timer mode selection.

Values:

enumerator kLPTMR_TimerModeTimeCounter

Time Counter mode

enumerator kLPTMR_TimerModePulseCounter
Pulse Counter mode

enum _lptmr_prescaler_glitch_value
LPTMR prescaler/glitch filter values.

Values:

enumerator kLPTMR_Prescale_Glitch_0
Prescaler divide 2, glitch filter does not support this setting

enumerator kLPTMR_Prescale_Glitch_1
Prescaler divide 4, glitch filter 2

enumerator kLPTMR_Prescale_Glitch_2
Prescaler divide 8, glitch filter 4

enumerator kLPTMR_Prescale_Glitch_3
Prescaler divide 16, glitch filter 8

enumerator kLPTMR_Prescale_Glitch_4
Prescaler divide 32, glitch filter 16

enumerator kLPTMR_Prescale_Glitch_5
Prescaler divide 64, glitch filter 32

enumerator kLPTMR_Prescale_Glitch_6
Prescaler divide 128, glitch filter 64

enumerator kLPTMR_Prescale_Glitch_7
Prescaler divide 256, glitch filter 128

enumerator kLPTMR_Prescale_Glitch_8
Prescaler divide 512, glitch filter 256

enumerator kLPTMR_Prescale_Glitch_9
Prescaler divide 1024, glitch filter 512

enumerator kLPTMR_Prescale_Glitch_10
Prescaler divide 2048 glitch filter 1024

enumerator kLPTMR_Prescale_Glitch_11
Prescaler divide 4096, glitch filter 2048

enumerator kLPTMR_Prescale_Glitch_12
Prescaler divide 8192, glitch filter 4096

enumerator kLPTMR_Prescale_Glitch_13
Prescaler divide 16384, glitch filter 8192

enumerator kLPTMR_Prescale_Glitch_14
Prescaler divide 32768, glitch filter 16384

enumerator kLPTMR_Prescale_Glitch_15
Prescaler divide 65536, glitch filter 32768

enum _lptmr_prescaler_clock_select
LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

Values:

enum `_lptmr_interrupt_enable`

List of the LPTMR interrupts.

Values:

enumerator `kLPTMR_TimerInterruptEnable`

Timer interrupt enable

enum `_lptmr_status_flags`

List of the LPTMR status flags.

Values:

enumerator `kLPTMR_TimerCompareFlag`

Timer compare flag

typedef enum `_lptmr_pin_select` `lptmr_pin_select_t`

LPTMR pin selection used in pulse counter mode.

typedef enum `_lptmr_pin_polarity` `lptmr_pin_polarity_t`

LPTMR pin polarity used in pulse counter mode.

typedef enum `_lptmr_timer_mode` `lptmr_timer_mode_t`

LPTMR timer mode selection.

typedef enum `_lptmr_prescaler_glitch_value` `lptmr_prescaler_glitch_value_t`

LPTMR prescaler/glitch filter values.

typedef enum `_lptmr_prescaler_clock_select` `lptmr_prescaler_clock_select_t`

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

typedef enum `_lptmr_interrupt_enable` `lptmr_interrupt_enable_t`

List of the LPTMR interrupts.

typedef enum `_lptmr_status_flags` `lptmr_status_flags_t`

List of the LPTMR status flags.

typedef struct `_lptmr_config` `lptmr_config_t`

LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the `LPTMR_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

static inline void `LPTMR_EnableTimerDMA(LPTMR_Type *base, bool enable)`

Enable or disable timer DMA request.

Parameters

- `base` – base LPTMR peripheral base address
- `enable` – Switcher of timer DMA feature. “true” means to enable, “false” means to disable.

struct `_lptmr_config`

`#include <fsl_lptmr.h>` LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the `LPTMR_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

Public Members

lptmr_timer_mode_t timerMode

Time counter mode or pulse counter mode

lptmr_pin_select_t pinSelect

LPTMR pulse input pin select; used only in pulse counter mode

lptmr_pin_polarity_t pinPolarity

LPTMR pulse input pin polarity; used only in pulse counter mode

bool enableFreeRunning

True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set

bool bypassPrescaler

True: bypass prescaler; false: use clock from prescaler

lptmr_prescaler_clock_select_t prescalerClockSource

LPTMR clock source

lptmr_prescaler_glitch_value_t value

Prescaler or glitch filter value

2.50 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.51 LPUART Driver

static inline void LPUART_SoftwareReset(LPUART_Type *base)

Resets the LPUART using software.

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

- base – LPUART peripheral base address.

status_t LPUART_Init(LPUART_Type *base, const *lpuart_config_t* *config, uint32_t srcClock_Hz)

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefaultConfig() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
```

(continues on next page)

(continued from previous page)

```
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

status_t LPUART_Deinit(LPUART_Type *base)

Deinitializes a LPUART instance.

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

- base – LPUART peripheral base address.

Return values

- kStatus_Success – Deinit is success.
- kStatus_LPUART_Timeout – Timeout during deinit.

void LPUART_GetDefaultConfig(*lpuart_config_t* *config)

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus_Success – Set baudrate succeeded.

void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, 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 cleared or set by hardware are: *kLPUART_TxDataRegEmptyFlag*, *kLPUART_TransmissionCompleteFlag*, *kLPUART_RxDataRegFullFlag*, *kLPUART_RxActiveFlag*, *kLPUART_NoiseErrorFlag*, *kLPUART_ParityErrorFlag*, *kLPUART_TxFifoEmptyFlag*, *kLPUART_RxFifoEmptyFlag* Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- base – LPUART peripheral base address.
- mask – the status flags to be cleared. The user can use the enumerators in the *_lpuart_status_flag_t* to do the OR operation and get the mask.

Return values

- *kStatus_LPUART_FlagCannotClearManually* – The flag can't be cleared by this function but it is cleared automatically by hardware.
- *kStatus_Success* – Status in the mask are cleared.

Returns

0 succeed, others failed.

void LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the *_lpuart_interrupt_enable*. This examples shows how to enable TX empty interrupt and RX full interrupt:

```
LPUART_EnableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_lpuart_interrupt_enable`.

`void LPUART_DisableInterrupts(LPUART_Type *base, uint32_t mask)`

Disables LPUART interrupts according to a provided mask.

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpuart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
LPUART_DisableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to disable. Logical OR of `_lpuart_interrupt_enable`.

`uint32_t LPUART_GetEnabledInterrupts(LPUART_Type *base)`

Gets enabled LPUART interrupts.

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_lpuart_interrupt_enable`. To check a specific interrupt enable status, compare the return value with enumerators in `_lpuart_interrupt_enable`. For example, to check whether the TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);

if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in `_lpuart_interrupt_enable`.

`static inline uintptr_t LPUART_GetDataRegisterAddress(LPUART_Type *base)`

Gets the LPUART data register address.

This function returns the LPUART data register address, which is mainly used by the DMA/eDMA.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

static inline void LPUART_EnableTxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter DMA request.

This function enables or disables the transmit data register empty flag, STAT[TDRE], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver DMA.

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

uint32_t LPUART_GetInstance(LPUART_Type *base)

Get the LPUART instance from peripheral base address.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART instance.

static inline void LPUART_EnableTx(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter.

This function enables or disables the LPUART transmitter.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRx(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver.

This function enables or disables the LPUART receiver.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_WriteByte(LPUART_Type *base, uint8_t data)

Writes to the transmitter register.

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

- base – LPUART peripheral base address.
- data – Data write to the TX register.

static inline uint8_t LPUART_ReadByte(LPUART_Type *base)

Reads the receiver register.

This function reads data from the receiver register directly. The upper layer must ensure that the receiver register is full or that the RX FIFO has data before calling this function.

Parameters

- base – LPUART peripheral base address.

Returns

Data read from data register.

static inline uint8_t LPUART_GetRxFifoCount(LPUART_Type *base)

Gets the rx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

rx FIFO data count.

static inline uint8_t LPUART_GetTxFifoCount(LPUART_Type *base)

Gets the tx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

tx FIFO data count.

void LPUART_SendAddress(LPUART_Type *base, uint8_t address)

Transmit an address frame in 9-bit data mode.

Parameters

- base – LPUART peripheral base address.
- address – LPUART slave address.

status_t LPUART_WriteBlocking(LPUART_Type *base, const uint8_t *data, size_t length)

Writes to the transmitter register using a blocking method.

This function polls the transmitter register, first waits for the register to be empty or TX FIFO to have room, and writes data to the transmitter buffer, then waits for the 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 TX register.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.
- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t LPUART_ReadBlocking16bit(LPUART_Type *base, uint16_t *data, size_t length)

Reads the receiver data register in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data by 16bit, only 10bit is valid.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.

- `kStatus_LPUART_ParityError` – Parity error happened while receiving data.
- `kStatus_LPUART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

```
void LPUART__TransferCreateHandle(LPUART_Type *base, lpuart_handle_t *handle,  
                                lpuart_transfer_callback_t callback, void *userData)
```

Initializes the LPUART handle.

This function initializes the LPUART handle, which can be used for other LPUART transactional APIs. Usually, for a specified LPUART instance, call this API once to get the initialized handle.

The LPUART driver supports the “background” receiving, which means that user can set up an RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn’t call the `LPUART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as `ringBuffer`.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `callback` – Callback function.
- `userData` – User data.

```
status_t LPUART__TransferSendNonBlocking(LPUART_Type *base, lpuart_handle_t *handle,  
                                         lpuart_transfer_t *xfer)
```

Transmits a buffer of data using the interrupt method.

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the `kStatus_LPUART_TxIdle` as status parameter.

Note: The `kStatus_LPUART_TxIdle` is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the TX, check the `kLPUART_TransmissionCompleteFlag` to ensure that the transmit is finished.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART transfer structure, see `lpuart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_LPUART_TxBusy` – Previous transmission still not finished, data not all written to the TX register.
- `kStatus_InvalidArgument` – Invalid argument.

```
void LPUART__TransferStartRingBuffer(LPUART_Type *base, lpuart_handle_t *handle, uint8_t  
                                     *ringBuffer, size_t ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the `UART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `ringBuffer` – Start address of ring buffer for background receiving. Pass `NULL` to disable the ring buffer.
- `ringBufferSize` – size of the ring buffer.

`void LPUART__TransferStopRingBuffer(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`size_t LPUART__TransferGetRxRingBufferLength(LPUART_Type *base, lpuart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

Returns

Length of received data in RX ring buffer.

`void LPUART__TransferAbortSend(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are not sent out.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t LPUART__TransferGetSendCount(LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by an interrupt method.

Parameters

- `base` – LPUART peripheral base address.

- `handle` – LPUART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`status_t` LPUART_TransferReceiveNonBlocking(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_t* *xfer, `size_t` *receivedBytes)

Receives a buffer of data using the interrupt method.

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter `kStatus_UART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to `xfer->data`, which returns with the parameter `receivedBytes` set to 5. For the remaining 5 bytes, the newly arrived data is saved from `xfer->data[5]`. When 5 bytes are received, the LPUART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART transfer structure, see `uart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into the transmit queue.
- `kStatus_LPUART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void` LPUART_TransferAbortReceive(LPUART_Type *base, *lpuart_handle_t* *handle)

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to find out how many bytes not received yet.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t` LPUART_TransferGetReceiveCount(LPUART_Type *base, *lpuart_handle_t* *handle, `uint32_t` *count)

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – LPUART peripheral base address.

- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

void LPUART_TransferHandleIRQ(LPUART_Type *base, void *irqHandle)
LPUART IRQ handle function.

This function handles the LPUART transmit and receive IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_TransferHandleErrorIRQ(LPUART_Type *base, void *irqHandle)
LPUART Error IRQ handle function.

This function handles the LPUART error IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_DriverIRQHandler(uint32_t instance)
LPUART driver IRQ handler common entry.

This function provides the common IRQ request entry for LPUART.

Parameters

- instance – LPUART instance.

FSL_LPUART_DRIVER_VERSION
LPUART driver version.

Error codes for the LPUART driver.

Values:

enumerator kStatus_LPUART_TxBusy
TX busy

enumerator kStatus_LPUART_RxBusy
RX busy

enumerator kStatus_LPUART_TxIdle
LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle
LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge
TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge
RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually

Some flag can't manually clear

enumerator kStatus_LPUART_Error

Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun

LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun

LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError

LPUART noise error.

enumerator kStatus_LPUART_FramingError

LPUART framing error.

enumerator kStatus_LPUART_ParityError

LPUART parity error.

enumerator kStatus_LPUART_BaudrateNotSupport

Baudrate is not support in current clock source

enumerator kStatus_LPUART_IdleLineDetected

IDLE flag.

enumerator kStatus_LPUART_Timeout

LPUART times out.

enum _lpuart_parity_mode

LPUART parity mode.

Values:

enumerator kLPUART_ParityDisabled

Parity disabled

enumerator kLPUART_ParityEven

Parity enabled, type even, bit setting: PE | PT = 10

enumerator kLPUART_ParityOdd

Parity enabled, type odd, bit setting: PE | PT = 11

enum _lpuart_data_bits

LPUART data bits count.

Values:

enumerator kLPUART_EightDataBits

Eight data bit

enumerator kLPUART_SevenDataBits

Seven data bit

enum _lpuart_stop_bit_count

LPUART stop bit count.

Values:

enumerator kLPUART_OneStopBit

One stop bit

enumerator kLPUART_TwoStopBit
Two stop bits

enum _lpuart_transmit_cts_source
LPUART transmit CTS source.

Values:

enumerator kLPUART_CtsSourcePin
CTS resource is the LPUART_CTS pin.

enumerator kLPUART_CtsSourceMatchResult
CTS resource is the match result.

enum _lpuart_transmit_cts_config
LPUART transmit CTS configure.

Values:

enumerator kLPUART_CtsSampleAtStart
CTS input is sampled at the start of each character.

enumerator kLPUART_CtsSampleAtIdle
CTS input is sampled when the transmitter is idle

enum _lpuart_idle_type_select
LPUART idle flag type defines when the receiver starts counting.

Values:

enumerator kLPUART_IdleTypeStartBit
Start counting after a valid start bit.

enumerator kLPUART_IdleTypeStopBit
Start counting after a stop bit.

enum _lpuart_idle_config
LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

Values:

enumerator kLPUART_IdleCharacter1
the number of idle characters.

enumerator kLPUART_IdleCharacter2
the number of idle characters.

enumerator kLPUART_IdleCharacter4
the number of idle characters.

enumerator kLPUART_IdleCharacter8
the number of idle characters.

enumerator kLPUART_IdleCharacter16
the number of idle characters.

enumerator kLPUART_IdleCharacter32
the number of idle characters.

enumerator kLPUART_IdleCharacter64
the number of idle characters.

enumerator kLPUART_IdleCharacter128
the number of idle characters.

enum _lpuart_interrupt_enable

LPUART interrupt configuration structure, default settings all disabled.

This structure contains the settings for all LPUART interrupt configurations.

Values:

enumerator kLPUART_LinBreakInterruptEnable
LIN break detect. bit 7

enumerator kLPUART_RxActiveEdgeInterruptEnable
Receive Active Edge. bit 6

enumerator kLPUART_TxDataRegEmptyInterruptEnable
Transmit data register empty. bit 23

enumerator kLPUART_TransmissionCompleteInterruptEnable
Transmission complete. bit 22

enumerator kLPUART_RxDataRegFullInterruptEnable
Receiver data register full. bit 21

enumerator kLPUART_IdleLineInterruptEnable
Idle line. bit 20

enumerator kLPUART_RxOverrunInterruptEnable
Receiver Overrun. bit 27

enumerator kLPUART_NoiseErrorInterruptEnable
Noise error flag. bit 26

enumerator kLPUART_FramingErrorInterruptEnable
Framing error flag. bit 25

enumerator kLPUART_ParityErrorInterruptEnable
Parity error flag. bit 24

enumerator kLPUART_Match1InterruptEnable
Parity error flag. bit 15

enumerator kLPUART_Match2InterruptEnable
Parity error flag. bit 14

enumerator kLPUART_TxFifoOverflowInterruptEnable
Transmit FIFO Overflow. bit 9

enumerator kLPUART_RxFifoUnderflowInterruptEnable
Receive FIFO Underflow. bit 8

enumerator kLPUART_AllInterruptEnable

enum _lpuart_flags

LPUART status flags.

This provides constants for the LPUART status flags for use in the LPUART functions.

Values:

enumerator kLPUART_TxDataRegEmptyFlag
Transmit data register empty flag, sets when transmit buffer is empty. bit 23

enumerator `kLPUART_TransmissionCompleteFlag`

Transmission complete flag, sets when transmission activity complete. bit 22

enumerator `kLPUART_RxDataRegFullFlag`

Receive data register full flag, sets when the receive data buffer is full. bit 21

enumerator `kLPUART_IdleLineFlag`

Idle line detect flag, sets when idle line detected. bit 20

enumerator `kLPUART_RxOverrunFlag`

Receive Overrun, sets when new data is received before data is read from receive register. bit 19

enumerator `kLPUART_NoiseErrorFlag`

Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets. bit 18

enumerator `kLPUART_FramingErrorFlag`

Frame error flag, sets if logic 0 was detected where stop bit expected. bit 17

enumerator `kLPUART_ParityErrorFlag`

If parity enabled, sets upon parity error detection. bit 16

enumerator `kLPUART_LinBreakFlag`

LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled. bit 31

enumerator `kLPUART_RxActiveEdgeFlag`

Receive pin active edge interrupt flag, sets when active edge detected. bit 30

enumerator `kLPUART_RxActiveFlag`

Receiver Active Flag (RAF), sets at beginning of valid start. bit 24

enumerator `kLPUART_DataMatch1Flag`

The next character to be read from LPUART_DATA matches MA1. bit 15

enumerator `kLPUART_DataMatch2Flag`

The next character to be read from LPUART_DATA matches MA2. bit 14

enumerator `kLPUART_TxFifoEmptyFlag`

TXEMPT bit, sets if transmit buffer is empty. bit 7

enumerator `kLPUART_RxFifoEmptyFlag`

RXEMPT bit, sets if receive buffer is empty. bit 6

enumerator `kLPUART_TxFifoOverflowFlag`

TXOF bit, sets if transmit buffer overflow occurred. bit 1

enumerator `kLPUART_RxFifoUnderflowFlag`

RXUF bit, sets if receive buffer underflow occurred. bit 0

enumerator `kLPUART_AllClearFlags`

enumerator `kLPUART_AllFlags`

typedef enum *lpuart_parity_mode* lpuart_parity_mode_t

LPUART parity mode.

typedef enum *lpuart_data_bits* lpuart_data_bits_t

LPUART data bits count.

typedef enum *lpuart_stop_bit_count* lpuart_stop_bit_count_t

LPUART stop bit count.

typedef enum *_lpuart_transmit_cts_source* lpuart_transmit_cts_source_t
LPUART transmit CTS source.

typedef enum *_lpuart_transmit_cts_config* lpuart_transmit_cts_config_t
LPUART transmit CTS configure.

typedef enum *_lpuart_idle_type_select* lpuart_idle_type_select_t
LPUART idle flag type defines when the receiver starts counting.

typedef enum *_lpuart_idle_config* lpuart_idle_config_t
LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

typedef struct *_lpuart_config* lpuart_config_t
LPUART configuration structure.

typedef struct *_lpuart_transfer* lpuart_transfer_t
LPUART transfer structure.

typedef struct *_lpuart_handle* lpuart_handle_t

typedef void (*lpuart_transfer_callback_t)(LPUART_Type *base, *lpuart_handle_t* *handle, *status_t* status, void *userData)
LPUART transfer callback function.

typedef void (*lpuart_isr_t)(LPUART_Type *base, void *handle)

void *s_lpuartHandle[]

const IRQn_Type s_lpuartTxIRQ[]

lpuart_isr_t s_lpuartIsr[]

UART_RETRY_TIMES
Retry times for waiting flag.

struct _lpuart_config
#include <fsl_lpuart.h> LPUART configuration structure.

Public Members

uint32_t baudRate_Bps
LPUART baud rate

lpuart_parity_mode_t parityMode
Parity mode, disabled (default), even, odd

lpuart_data_bits_t dataBitsCount
Data bits count, eight (default), seven

bool isMsb
Data bits order, LSB (default), MSB

lpuart_stop_bit_count_t stopBitCount
Number of stop bits, 1 stop bit (default) or 2 stop bits

uint8_t txFifoWatermark
TX FIFO watermark

uint8_t rxFifoWatermark
RX FIFO watermark

`bool enableRxRTS`
RX RTS enable

`bool enableTxCTS`
TX CTS enable

`lpuart_transmit_cts_source_t txCtsSource`
TX CTS source

`lpuart_transmit_cts_config_t txCtsConfig`
TX CTS configure

`uint8_t rtsWatermark`
RTS watermark

`lpuart_idle_type_select_t rxIdleType`
RX IDLE type.

`lpuart_idle_config_t rxIdleConfig`
RX IDLE configuration.

`bool enableTx`
Enable TX

`bool enableRx`
Enable RX

`bool swapTxdRxd`
Swap TXD and RXD pins

`struct __lpuart_transfer`
#include <fsl_lpuart.h> LPUART transfer structure.

Public Members

`size_t dataSize`
The byte count to be transfer.

`struct __lpuart_handle`
#include <fsl_lpuart.h> LPUART handle structure.

Public Members

`volatile size_t txDataSize`
Size of the remaining data to send.

`size_t txDataSizeAll`
Size of the data to send out.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t rxDataSizeAll`
Size of the data to receive.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail

Index for the user to get data from the ring buffer.

lpuart_transfer_callback_t callback

Callback function.

void *userData

LPUART callback function parameter.

volatile uint8_t txState

TX transfer state.

volatile uint8_t rxState

RX transfer state.

bool isSevenDataBits

Seven data bits flag.

bool is16bitData

16bit data bits flag, only used for 9bit or 10bit data

union ____unnamed58____

Public Members

uint8_t *data

The buffer of data to be transfer.

uint8_t *rxData

The buffer to receive data.

uint16_t *rxData16

The buffer to receive data.

const uint8_t *txData

The buffer of data to be sent.

const uint16_t *txData16

The buffer of data to be sent.

union ____unnamed60____

Public Members

const uint8_t *volatile txData

Address of remaining data to send.

const uint16_t *volatile txData16

Address of remaining data to send.

union ____unnamed62____

Public Members

uint8_t *volatile rxData

Address of remaining data to receive.

uint16_t *volatile rxData16

Address of remaining data to receive.

union ____unnamed64____

Public Members

uint8_t *rxRingBuffer

Start address of the receiver ring buffer.

uint16_t *rxRingBuffer16

Start address of the receiver ring buffer.

2.52 LPUART eDMA Driver

```
void LPUART__TransferCreateHandleEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                                     lpuart_edma_transfer_callback_t callback, void  
                                     *userData, edma_handle_t *txEdmaHandle,  
                                     edma_handle_t *rxEdmaHandle)
```

Initializes the LPUART handle which is used in transactional functions.

Note: This function disables all LPUART interrupts.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to lpuart_edma_handle_t structure.
- callback – Callback function.
- userData – User data.
- txEdmaHandle – User requested DMA handle for TX DMA transfer.
- rxEdmaHandle – User requested DMA handle for RX DMA transfer.

```
status_t LPUART__SendEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                          lpuart_transfer_t *xfer)
```

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART eDMA transfer structure. See lpuart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_LPUART_TxBusy – Previous transfer on going.
- kStatus_InvalidArgument – Invalid argument.

```
status_t LPUART__ReceiveEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                             lpuart_transfer_t *xfer)
```

Receives data using eDMA.

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.
- xfer – LPUART eDMA transfer structure, see `lpuart_transfer_t`.

Return values

- `kStatus__Success` – if succeed, others fail.
- `kStatus__LPUART_RxBusy` – Previous transfer ongoing.
- `kStatus__InvalidArgument` – Invalid argument.

`void LPUART__TransferAbortSendEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle)`
Aborts the sent data using eDMA.

This function aborts the sent data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.

`void LPUART__TransferAbortReceiveEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle)`
Aborts the received data using eDMA.

This function aborts the received data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.

`status_t LPUART__TransferGetSendCountEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)`

Gets the number of bytes written to the LPUART TX register.

This function gets the number of bytes written to the LPUART TX register by DMA.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Send bytes count.

Return values

- `kStatus__NoTransferInProgress` – No send in progress.
- `kStatus__InvalidArgument` – Parameter is invalid.
- `kStatus__Success` – Get successfully through the parameter count;

`status_t LPUART__TransferGetReceiveCountEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)`

Gets the number of received bytes.

This function gets the number of received bytes.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

`void LPUART_TransferEdmaHandleIRQ(LPUART_Type *base, void *lpuartEdmaHandle)`

LPUART eDMA IRQ handle function.

This function handles the LPUART tx complete IRQ request and invoke user callback. It is not set to static so that it can be used in user application.

Note: This function is used as default IRQ handler by double weak mechanism. If user's specific IRQ handler is implemented, make sure this function is invoked in the handler.

Parameters

- `base` – LPUART peripheral base address.
- `lpuartEdmaHandle` – LPUART handle pointer.

`FSL_LPUART_EDMA_DRIVER_VERSION`

LPUART EDMA driver version.

`typedef struct _lpuart_edma_handle lpuart_edma_handle_t`

`typedef void (*lpuart_edma_transfer_callback_t)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)`

LPUART transfer callback function.

`struct _lpuart_edma_handle`

`#include <fsl_lpuart_edma.h>` LPUART eDMA handle.

Public Members

`lpuart_edma_transfer_callback_t` callback

Callback function.

`void *userData`

LPUART callback function parameter.

`size_t rxDataSizeAll`

Size of the data to receive.

`size_t txDataSizeAll`

Size of the data to send out.

`edma_handle_t *txEdmaHandle`

The eDMA TX channel used.

`edma_handle_t *rxEdmaHandle`

The eDMA RX channel used.

`uint8_t nbytes`

eDMA minor byte transfer count initially configured.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.53 LTC: LP Trusted Cryptography

FSL_LTC_DRIVER_VERSION

LTC driver version. Version 2.0.18.

Current version: 2.0.18

Change log:

- Version 2.0.1
 - fixed warning during g++ compilation
- Version 2.0.2
 - fixed [KPSDK-10932][LTC][SHA] LTC_HASH() blocks indefinitely when message size exceeds 4080 bytes
- Version 2.0.3
 - fixed LTC_PKHA_CompareBigNum() in case an integer argument is an array of all zeroes
- Version 2.0.4
 - constant LTC_PKHA_CompareBigNum() processing time
- Version 2.0.5
 - Fix MISRA issues
- Version 2.0.6
 - fixed [KPSDK-23603][LTC] AES Decrypt in ECB and CBC modes fail when ciphertext size > 0xff0 bytes
- Version 2.0.7
 - Fix MISRA-2012 issues
- Version 2.0.8
 - Fix Coverity issues
- Version 2.0.9
 - Fix sign-compare warning in ltc_set_context and in ltc_get_context
- Version 2.0.10
 - Fix MISRA-2012 issues
- Version 2.0.11
 - Fix MISRA-2012 issues
- Version 2.0.12
 - Fix AES Decrypt in CBC modes fail when used kLTC_DecryptKey.
- Version 2.0.13
 - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_Init function.
- Version 2.0.14
 - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_Deinit function.
- Version 2.0.15
 - Fix MISRA-2012 issues

- Version 2.0.16
 - Fix uninitialized GCC warning in LTC_AES_GenerateDecryptKey()
- Version 2.0.17
 - Fix CMAC for payloads over one block, and if BRIC is present on the device, remove XCBC and “decrypt key” functionality
- Version 2.0.18
 - Fix CERT INT30-C and INT31-C compliance

`void LTC_Init(LTC_Type *base)`

Initializes the LTC driver. This function initializes the LTC driver.

Parameters

- `base` – LTC peripheral base address

`void LTC_Deinit(LTC_Type *base)`

Deinitializes the LTC driver. This function deinitializes the LTC driver.

Parameters

- `base` – LTC peripheral base address

`void LTC_SetDpaMaskSeed(LTC_Type *base, uint32_t mask)`

Sets the DPA Mask Seed register.

The DPA Mask Seed register reseeds the mask that provides resistance against DPA (differential power analysis) attacks on AES or DES keys.

Differential Power Analysis Mask (DPA) resistance uses a randomly changing mask that introduces “noise” into the power consumed by the AES or DES. This reduces the signal-to-noise ratio that differential power analysis attacks use to “guess” bits of the key. This randomly changing mask should be seeded at POR, and continues to provide DPA resistance from that point on. However, to provide even more DPA protection it is recommended that the DPA mask be reseeded after every 50,000 blocks have been processed. At that time, software can opt to write a new seed (preferably obtained from an RNG) into the DPA Mask Seed register (DPAMS), or software can opt to provide the new seed earlier or later, or not at all. DPA resistance continues even if the DPA mask is never reseeded.

Parameters

- `base` – LTC peripheral base address
- `mask` – The DPA mask seed.

2.54 LTC AES driver

`enum _ltc_aes_key_t`

Type of AES key for ECB and CBC decrypt operations.

Values:

enumerator `kLTC_EncryptKey`

Input key is an encrypt key

`typedef enum _ltc_aes_key_t ltc_aes_key_t`

Type of AES key for ECB and CBC decrypt operations.

```
status_t LTC_AES_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t *key, uint32_t keySize)
```

Encrypts AES using the ECB block mode.

Encrypts AES using the ECB block mode.

Parameters

- *base* – LTC peripheral base address
- *plaintext* – Input plain text to encrypt
- *ciphertext* – **[out]** Output cipher text
- *size* – Size of input and output data in bytes. Must be multiple of 16 bytes.
- *key* – Input key to use for encryption
- *keySize* – Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

```
status_t LTC_AES_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t *key, uint32_t keySize, ltc_aes_key_t  
                             keyType)
```

Decrypts AES using ECB block mode.

Decrypts AES using ECB block mode.

Parameters

- *base* – LTC peripheral base address
- *ciphertext* – Input cipher text to decrypt
- *plaintext* – **[out]** Output plain text
- *size* – Size of input and output data in bytes. Must be multiple of 16 bytes.
- *key* – Input key.
- *keySize* – Size of the input key, in bytes. Must be 16, 24, or 32.
- *keyType* – Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.)

Returns

Status from decrypt operation

```
status_t LTC_AES_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[16], const uint8_t *key, uint32_t  
                             keySize)
```

Encrypts AES using CBC block mode.

Parameters

- *base* – LTC peripheral base address
- *plaintext* – Input plain text to encrypt
- *ciphertext* – **[out]** Output cipher text
- *size* – Size of input and output data in bytes. Must be multiple of 16 bytes.
- *iv* – Input initial vector to combine with the first input block.
- *key* – Input key to use for encryption
- *keySize* – Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

```
status_t LTC_AES_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[16], const uint8_t *key, uint32_t  
                             keySize, ltc_aes_key_t keyType)
```

Decrypts AES using CBC block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.
- key – Input key to use for decryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- keyType – Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.)

Returns

Status from decrypt operation

```
status_t LTC_AES_CryptCtr(LTC_Type *base, const uint8_t *input, uint8_t *output, uint32_t  
                           size, uint8_t counter[16U], const uint8_t *key, uint32_t keySize,  
                           uint8_t counterlast[16U], uint32_t *szLeft)
```

Encrypts or decrypts AES using CTR block mode.

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

- base – LTC peripheral base address
- input – Input data for CTR block mode
- output – **[out]** Output data for CTR block mode
- size – Size of input and output data in bytes
- counter – **[inout]** Input counter (updates on return)
- key – Input key to use for forward AES cipher
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- counterlast – **[out]** Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.
- szLeft – **[out]** Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns

Status from encrypt operation

```
status_t LTC_AES_EncryptTagGcm(LTC_Type *base, const uint8_t *plaintext, uint8_t  
                                *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize,  
                                const uint8_t *aad, uint32_t aadSize, const uint8_t *key,  
                                uint32_t keySize, uint8_t *tag, uint32_t tagSize)
```

Encrypts AES and tags using GCM block mode.

Encrypts AES and optionally tags using GCM block mode. If plaintext is NULL, only the GHASH is calculated and output in the 'tag' field.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text.
- size – Size of input and output data in bytes
- iv – Input initial vector
- ivSize – Size of the IV
- aad – Input additional authentication data
- aadSize – Input size in bytes of AAD
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – **[out]** Output hash tag. Set to NULL to skip tag processing.
- tagSize – Input size of the tag to generate, in bytes. Must be 4,8,12,13,14,15 or 16.

Returns

Status from encrypt operation

```
status_t LTC_AES_DecryptTagGcm(LTC_Type *base, const uint8_t *ciphertext, uint8_t  
                                *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize,  
                                const uint8_t *aad, uint32_t aadSize, const uint8_t *key,  
                                uint32_t keySize, const uint8_t *tag, uint32_t tagSize)
```

Decrypts AES and authenticates using GCM block mode.

Decrypts AES and optionally authenticates using GCM block mode. If ciphertext is NULL, only the GHASH is calculated and compared with the received GHASH in 'tag' field.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text.
- size – Size of input and output data in bytes
- iv – Input initial vector
- ivSize – Size of the IV
- aad – Input additional authentication data
- aadSize – Input size in bytes of AAD
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – Input hash tag to compare. Set to NULL to skip tag processing.
- tagSize – Input size of the tag, in bytes. Must be 4, 8, 12, 13, 14, 15, or 16.

Returns

Status from decrypt operation

```
status_t LTC_AES_EncryptTagCcm(LTC_Type *base, const uint8_t *plaintext, uint8_t  
                                *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize,  
                                const uint8_t *aad, uint32_t aadSize, const uint8_t *key,  
                                uint32_t keySize, uint8_t *tag, uint32_t tagSize)
```

Encrypts AES and tags using CCM block mode.

Encrypts AES and optionally tags using CCM block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text.
- size – Size of input and output data in bytes. Zero means authentication only.
- iv – Nonce
- ivSize – Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
- aad – Input additional authentication data. Can be NULL if aadSize is zero.
- aadSize – Input size in bytes of AAD. Zero means data mode only (authentication skipped).
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – **[out]** Generated output tag. Set to NULL to skip tag processing.
- tagSize – Input size of the tag to generate, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns

Status from encrypt operation

```
status_t LTC_AES_DecryptTagCcm(LTC_Type *base, const uint8_t *ciphertext, uint8_t  
                                *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize,  
                                const uint8_t *aad, uint32_t aadSize, const uint8_t *key,  
                                uint32_t keySize, const uint8_t *tag, uint32_t tagSize)
```

Decrypts AES and authenticates using CCM block mode.

Decrypts AES and optionally authenticates using CCM block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text.
- size – Size of input and output data in bytes. Zero means authentication only.
- iv – Nonce
- ivSize – Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
- aad – Input additional authentication data. Can be NULL if aadSize is zero.
- aadSize – Input size in bytes of AAD. Zero means data mode only (authentication skipped).
- key – Input key to use for decryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.

- tag – Received tag. Set to NULL to skip tag processing.
- tagSize – Input size of the received tag to compare with the computed tag, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns

Status from decrypt operation

LTC_AES_BLOCK_SIZE

AES block size in bytes

LTC_AES_IV_SIZE

AES Input Vector size in bytes

LTC_KEY_REGISTER_READABLE

LTC_AES_DecryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)

AES CTR decrypt is mapped to the AES CTR generic operation

LTC_AES_EncryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)

AES CTR encrypt is mapped to the AES CTR generic operation

2.55 LTC DES driver

status_t LTC_DES_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key[8])

Encrypts DES using ECB block mode.

Encrypts DES using ECB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key – Input key to use for encryption

Returns

Status from encrypt/decrypt operation

status_t LTC_DES_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key[8])

Decrypts DES using ECB block mode.

Decrypts DES using ECB block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key – Input key to use for decryption

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Encrypts DES using CBC block mode.

Encrypts DES using CBC block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key – Input key to use for encryption

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Decrypts DES using CBC block mode.

Decrypts DES using CBC block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key – Input key to use for decryption

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Encrypts DES using CFB block mode.

Encrypts DES using CFB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- size – Size of input data in bytes
- iv – Input initial block.
- key – Input key to use for encryption
- ciphertext – **[out]** Output ciphertext

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Decrypts DES using CFB block mode.

Decrypts DES using CFB block mode.

Parameters

- *base* – LTC peripheral base address
- *ciphertext* – Input ciphertext to decrypt
- *plaintext* – **[out]** Output plaintext
- *size* – Size of input and output data in bytes
- *iv* – Input initial block.
- *key* – Input key to use for decryption

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Encrypts DES using OFB block mode.

Encrypts DES using OFB block mode.

Parameters

- *base* – LTC peripheral base address
- *plaintext* – Input plaintext to encrypt
- *ciphertext* – **[out]** Output ciphertext
- *size* – Size of input and output data in bytes
- *iv* – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- *key* – Input key to use for encryption

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key[8])
```

Decrypts DES using OFB block mode.

Decrypts DES using OFB block mode.

Parameters

- *base* – LTC peripheral base address
- *ciphertext* – Input ciphertext to decrypt
- *plaintext* – **[out]** Output plaintext
- *size* – Size of input and output data in bytes. Must be multiple of 8 bytes.
- *iv* – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- *key* – Input key to use for decryption

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using ECB block mode with two keys.

Encrypts triple DES using ECB block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8])

Decrypts triple DES using ECB block mode with two keys.

Decrypts triple DES using ECB block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using CBC block mode with two keys.

Encrypts triple DES using CBC block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES2_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8])
```

Decrypts triple DES using CBC block mode with two keys.

Decrypts triple DES using CBC block mode with two keys.

Parameters

- `base` – LTC peripheral base address
- `ciphertext` – Input ciphertext to decrypt
- `plaintext` – **[out]** Output plaintext
- `size` – Size of input and output data in bytes
- `iv` – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- `key1` – First input key for key bundle
- `key2` – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES2_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8])
```

Encrypts triple DES using CFB block mode with two keys.

Encrypts triple DES using CFB block mode with two keys.

Parameters

- `base` – LTC peripheral base address
- `plaintext` – Input plaintext to encrypt
- `ciphertext` – **[out]** Output ciphertext
- `size` – Size of input and output data in bytes
- `iv` – Input initial block.
- `key1` – First input key for key bundle
- `key2` – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES2_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8])
```

Decrypts triple DES using CFB block mode with two keys.

Decrypts triple DES using CFB block mode with two keys.

Parameters

- `base` – LTC peripheral base address
- `ciphertext` – Input ciphertext to decrypt
- `plaintext` – **[out]** Output plaintext
- `size` – Size of input and output data in bytes
- `iv` – Input initial block.

- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES2_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8])
```

Encrypts triple DES using OFB block mode with two keys.

Encrypts triple DES using OFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES2_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8])
```

Decrypts triple DES using OFB block mode with two keys.

Decrypts triple DES using OFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t key1[8], const uint8_t key2[8], const  
                             uint8_t key3[8])
```

Encrypts triple DES using ECB block mode with three keys.

Encrypts triple DES using ECB block mode with three keys.

Parameters

- base – LTC peripheral base address

- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t key1[8], const uint8_t key2[8], const  
                             uint8_t key3[8])
```

Decrypts triple DES using ECB block mode with three keys.

Decrypts triple DES using ECB block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8], const uint8_t key3[8])
```

Encrypts triple DES using CBC block mode with three keys.

Encrypts triple DES using CBC block mode with three keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8], const uint8_t key3[8])
```

Decrypts triple DES using CBC block mode with three keys.

Decrypts triple DES using CBC block mode with three keys.

Parameters

- *base* – LTC peripheral base address
- *ciphertext* – Input ciphertext to decrypt
- *plaintext* – **[out]** Output plaintext
- *size* – Size of input and output data in bytes
- *iv* – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- *key1* – First input key for key bundle
- *key2* – Second input key for key bundle
- *key3* – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8], const uint8_t key3[8])
```

Encrypts triple DES using CFB block mode with three keys.

Encrypts triple DES using CFB block mode with three keys.

Parameters

- *base* – LTC peripheral base address
- *plaintext* – Input plaintext to encrypt
- *ciphertext* – **[out]** Output ciphertext
- *size* – Size of input and output data in bytes
- *iv* – Input initial block.
- *key1* – First input key for key bundle
- *key2* – Second input key for key bundle
- *key3* – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8], const uint8_t key3[8])
```

Decrypts triple DES using CFB block mode with three keys.

Decrypts triple DES using CFB block mode with three keys.

Parameters

- *base* – LTC peripheral base address
- *ciphertext* – Input ciphertext to decrypt
- *plaintext* – **[out]** Output plaintext

- size – Size of input data in bytes
- iv – Input initial block.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext,  
                             uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                             uint8_t key2[8], const uint8_t key3[8])
```

Encrypts triple DES using OFB block mode with three keys.

Encrypts triple DES using OFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

```
status_t LTC_DES3_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext,  
                              uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const  
                              uint8_t key2[8], const uint8_t key3[8])
```

Decrypts triple DES using OFB block mode with three keys.

Decrypts triple DES using OFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

LTC_DES_KEY_SIZE

LTC DES key size - 64 bits.

LTC_DES_IV_SIZE

LTC DES IV size - 8 bytes.

2.56 LTC HASH driver

enum _ltc_hash_algo_t

Supported cryptographic block cipher functions for HASH creation

Values:

enumerator kLTC_Cmac

CMAC (AES engine)

enumerator kLTC_Sha1

SHA_1 (MDHA engine)

enumerator kLTC_Sha224

SHA_224 (MDHA engine)

enumerator kLTC_Sha256

SHA_256 (MDHA engine)

typedef enum _ltc_hash_algo_t ltc_hash_algo_t

Supported cryptographic block cipher functions for HASH creation

typedef struct _ltc_hash_ctx_t ltc_hash_ctx_t

Storage type used to save hash context.

status_t LTC_HASH_Init(LTC_Type *base, ltc_hash_ctx_t *ctx, ltc_hash_algo_t algo, const uint8_t *key, uint32_t keySize)

Initialize HASH context.

This function initialize the HASH. Key shall be supplied if the underlying algorithm is AES XCBC-MAC or CMAC. Key shall be NULL if the underlying algorithm is SHA.

For XCBC-MAC, the key length must be 16. For CMAC, the key length can be the AES key lengths supported by AES engine. For MDHA the key length argument is ignored.

Parameters

- base – LTC peripheral base address
- ctx – **[out]** Output hash context
- algo – Underlying algorithm to use for hash computation.
- key – Input key (NULL if underlying algorithm is SHA)
- keySize – Size of input key in bytes

Returns

Status of initialization

status_t LTC_HASH_Update(ltc_hash_ctx_t *ctx, const uint8_t *input, uint32_t inputSize)

Add data to current HASH.

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed.

Parameters

- `ctx` – **[inout]** HASH context
- `input` – Input data
- `inputSize` – Size of input data in bytes

Returns

Status of the hash update operation

`status_t LTC_HASH_Finish(ltc_hash_ctx_t *ctx, uint8_t *output, uint32_t *outputSize)`
Finalize hashing.

Outputs the final hash and erases the context.

Parameters

- `ctx` – **[inout]** Input hash context
- `output` – **[out]** Output hash data
- `outputSize` – **[out]** Output parameter storing the size of the output hash in bytes

Returns

Status of the hash finish operation

`status_t LTC_HASH(LTC_Type *base, ltc_hash_algo_t algo, const uint8_t *input, uint32_t inputSize, const uint8_t *key, uint32_t keySize, uint8_t *output, uint32_t *outputSize)`

Create HASH on given data.

Perform the full keyed HASH in one function call.

Parameters

- `base` – LTC peripheral base address
- `algo` – Block cipher algorithm to use for CMAC creation
- `input` – Input data
- `inputSize` – Size of input data in bytes
- `key` – Input key
- `keySize` – Size of input key in bytes
- `output` – **[out]** Output hash data
- `outputSize` – **[out]** Output parameter storing the size of the output hash in bytes

Returns

Status of the one call hash operation.

`LTC_HASH_CTX_SIZE`

LTC HASH Context size.

`struct _ltc_hash_ctx_t`

`#include <fsl_ltc.h>` Storage type used to save hash context.

2.57 LTC PKHA driver

`enum _ltc_pkha_timing_t`

Use of timing equalized version of a PKHA function.

Values:

enumerator kLTC_PKHA_NoTimingEqualized

Normal version of a PKHA operation

enumerator kLTC_PKHA_TimingEqualized

Timing-equalized version of a PKHA operation

enum _ltc_pkha_f2m_t

Integer vs binary polynomial arithmetic selection.

Values:

enumerator kLTC_PKHA_IntegerArith

Use integer arithmetic

enumerator kLTC_PKHA_F2mArith

Use binary polynomial arithmetic

enum _ltc_pkha_montgomery_form_t

Montgomery or normal PKHA input format.

Values:

enumerator kLTC_PKHA_NormalValue

PKHA number is normal integer

enumerator kLTC_PKHA_MontgomeryFormat

PKHA number is in montgomery format

typedef struct _ltc_pkha_ecc_point_t ltc_pkha_ecc_point_t

PKHA ECC point structure

typedef enum _ltc_pkha_timing_t ltc_pkha_timing_t

Use of timing equalized version of a PKHA function.

typedef enum _ltc_pkha_f2m_t ltc_pkha_f2m_t

Integer vs binary polynomial arithmetic selection.

typedef enum _ltc_pkha_montgomery_form_t ltc_pkha_montgomery_form_t

Montgomery or normal PKHA input format.

int LTC_PKHA_CompareBigNum(const uint8_t *a, size_t sizeA, const uint8_t *b, size_t sizeB)

Compare two PKHA big numbers.

Compare two PKHA big numbers. Return 1 for $a > b$, -1 for $a < b$ and 0 if they are same. PKHA big number is lsbyte first. Thus the comparison starts at msbyte which is the last member of tested arrays.

Parameters

- a – First integer represented as an array of bytes, lsbyte first.
- sizeA – Size in bytes of the first integer.
- b – Second integer represented as an array of bytes, lsbyte first.
- sizeB – Size in bytes of the second integer.

Returns

1 if $a > b$.

Returns

-1 if $a < b$.

Returns

0 if $a = b$.

```
status_t LTC_PKHA_NormalToMontgomery(LTC_Type *base, const uint8_t *N, uint16_t sizeN,  
                                       uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t  
                                       *sizeB, uint8_t *R2, uint16_t *sizeR2,  
                                       ltc_pkha_timing_t equalTime, ltc_pkha_f2m_t  
                                       arithType)
```

Converts from integer to Montgomery format.

This function computes $R2 \bmod N$ and optionally converts A or B into Montgomery format of A or B.

Parameters

- base – LTC peripheral base address
- N – modulus
- sizeN – size of N in bytes
- A – **[inout]** The first input in non-Montgomery format. Output Montgomery format of the first input.
- sizeA – **[inout]** pointer to size variable. On input it holds size of input A in bytes. On output it holds size of Montgomery format of A in bytes.
- B – **[inout]** Second input in non-Montgomery format. Output Montgomery format of the second input.
- sizeB – **[inout]** pointer to size variable. On input it holds size of input B in bytes. On output it holds size of Montgomery format of B in bytes.
- R2 – **[out]** Output Montgomery factor $R2 \bmod N$.
- sizeR2 – **[out]** pointer to size variable. On output it holds size of Montgomery factor $R2 \bmod N$ in bytes.
- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_MontgomeryToNormal(LTC_Type *base, const uint8_t *N, uint16_t sizeN,  
                                       uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t  
                                       *sizeB, ltc_pkha_timing_t equalTime, ltc_pkha_f2m_t  
                                       arithType)
```

Converts from Montgomery format to int.

This function converts Montgomery format of A or B into int A or B.

Parameters

- base – LTC peripheral base address
- N – modulus.
- sizeN – size of N modulus in bytes.
- A – **[inout]** Input first number in Montgomery format. Output is non-Montgomery format.
- sizeA – **[inout]** pointer to size variable. On input it holds size of the input A in bytes. On output it holds size of non-Montgomery A in bytes.
- B – **[inout]** Input first number in Montgomery format. Output is non-Montgomery format.
- sizeB – **[inout]** pointer to size variable. On input it holds size of the input B in bytes. On output it holds size of non-Montgomery B in bytes.

- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_ModAdd(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                        *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t
                        *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
```

Performs modular addition - $(A + B) \bmod N$.

This function performs modular addition of $(A + B) \bmod N$, with either integer or binary polynomial (F2m) inputs. In the F2m form, this function is equivalent to a bitwise XOR and it is functionally the same as subtraction.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus. For F2m operation this can be NULL, as N is ignored during F2m polynomial addition.
- sizeN – Size of N in bytes. This must be given for both integer and F2m polynomial additions.
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_ModSub1(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                        *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t
                        *result, uint16_t *resultSize)
```

Performs modular subtraction - $(A - B) \bmod N$.

This function performs modular subtraction of $(A - B) \bmod N$ with integer inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes

Returns

Operation status.

```
status_t LTC_PKHA_ModSub2(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                           *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t
                           *result, uint16_t *resultSize)
```

Performs modular subtraction - $(B - A) \bmod N$.

This function performs modular subtraction of $(B - A) \bmod N$, with integer inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes

Returns

Operation status.

```
status_t LTC_PKHA_ModMul(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                           *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t
                           *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType,
                           ltc_pkha_montgomery_form_t montIn,
                           ltc_pkha_montgomery_form_t montOut, ltc_pkha_timing_t
                           equalTime)
```

Performs modular multiplication - $(A \times B) \bmod N$.

This function performs modular multiplication with either integer or binary polynomial (F2m) inputs. It can optionally specify whether inputs and/or outputs will be in Montgomery form or not.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus.
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)
- montIn – Format of inputs
- montOut – Format of output
- equalTime – Run the function time equalized or no timing equalization. This argument is ignored for F2m modular multiplication.

Returns

Operation status.

```
status_t LTC_PKHA_ModExp(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                        *N, uint16_t sizeN, const uint8_t *E, uint16_t sizeE, uint8_t
                        *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType,
                        ltc_pkha_montgomery_form_t montIn, ltc_pkha_timing_t
                        equalTime)
```

Performs modular exponentiation - $(A^E) \bmod N$.

This function performs modular exponentiation with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- E – exponent
- sizeE – Size of E in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- montIn – Format of A input (normal or Montgomery)
- arithType – Type of arithmetic to perform (integer or F2m)
- equalTime – Run the function time equalized or no timing equalization.

Returns

Operation status.

```
status_t LTC_PKHA_ModRed(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t
                        *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize,
                        ltc_pkha_f2m_t arithType)
```

Performs modular reduction - $(A) \bmod N$.

This function performs modular reduction with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_ModInv(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N,
                        uint16_t sizeN, uint8_t *result, uint16_t *resultSize,
                        ltc_pkha_f2m_t arithType)
```

Performs modular inversion - $(A^{-1}) \bmod N$.

This function performs modular inversion with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_ModR2(LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *result,
                        uint16_t *resultSize, ltc_pkha_f2m_t arithType)
```

Computes integer Montgomery factor $R^2 \bmod N$.

This function computes a constant to assist in converting operands into the Montgomery residue system representation.

Parameters

- base – LTC peripheral base address
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_GCD(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N,
                     uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t
                     arithType)
```

Calculates the greatest common divisor - GCD (A, N).

This function calculates the greatest common divisor of two inputs with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first value (must be smaller than or equal to N)
- sizeA – Size of A in bytes
- N – second value (must be non-zero)
- sizeN – Size of N in bytes

- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

```
status_t LTC_PKHA_PrimalityTest(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const
                                uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN,
                                bool *res)
```

Executes Miller-Rabin primality test.

This function calculates whether or not a candidate prime number is likely to be a prime.

Parameters

- base – LTC peripheral base address
- A – initial random seed
- sizeA – Size of A in bytes
- B – number of trial runs
- sizeB – Size of B in bytes
- N – candidate prime integer
- sizeN – Size of N in bytes
- res – **[out]** True if the value is likely prime or false otherwise

Returns

Operation status.

```
status_t LTC_PKHA_ECC_PointAdd(LTC_Type *base, const ltc_pkha_ecc_point_t *A, const
                                ltc_pkha_ecc_point_t *B, const uint8_t *N, const uint8_t
                                *R2modN, const uint8_t *aCurveParam, const uint8_t
                                *bCurveParam, uint8_t size, ltc_pkha_f2m_t arithType,
                                ltc_pkha_ecc_point_t *result)
```

Adds elliptic curve points - A + B.

This function performs ECC point addition over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

- base – LTC peripheral base address
- A – Left-hand point
- B – Right-hand point
- N – Prime modulus of the field
- R2modN – NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (constant)
- size – Size in bytes of curve points and parameters
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

Returns

Operation status.

```
status_t LTC_PKHA_ECC_PointDouble(LTC_Type *base, const ltc_pkha_ecc_point_t *B, const
                                uint8_t *N, const uint8_t *aCurveParam, const uint8_t
                                *bCurveParam, uint8_t size, ltc_pkha_f2m_t arithType,
                                ltc_pkha_ecc_point_t *result)
```

Doubles elliptic curve points - $B + B$.

This function performs ECC point doubling over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

- base – LTC peripheral base address
- B – Point to double
- N – Prime modulus of the field
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (constant)
- size – Size in bytes of curve points and parameters
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

Returns

Operation status.

```
status_t LTC_PKHA_ECC_PointMul(LTC_Type *base, const ltc_pkha_ecc_point_t *A, const
                               uint8_t *E, uint8_t sizeE, const uint8_t *N, const uint8_t
                               *R2modN, const uint8_t *aCurveParam, const uint8_t
                               *bCurveParam, uint8_t size, ltc_pkha_timing_t equalTime,
                               ltc_pkha_f2m_t arithType, ltc_pkha_ecc_point_t *result,
                               bool *infinity)
```

Multiplies an elliptic curve point by a scalar - $E \times (A0, A1)$.

This function performs ECC point multiplication to multiply an ECC point by a scalar integer multiplier over a prime field (Fp) or a binary field (F2m).

Parameters

- base – LTC peripheral base address
- A – Point as multiplicand
- E – Scalar multiple
- sizeE – The size of E, in bytes
- N – Modulus, a prime number for the Fp field or Irreducible polynomial for F2m field.
- R2modN – NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (C parameter for operation over F2m).
- size – Size in bytes of curve points and parameters
- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

- infinity – **[out]** Output true if the result is point of infinity, and false otherwise. Writing of this output will be ignored if the argument is NULL.

Returns

Operation status.

```
struct __ltc_pkha_ecc_point_t  
#include <fsl_ltc.h> PKHA ECC point structure
```

Public Members

```
uint8_t *X  
    X coordinate (affine)  
uint8_t *Y  
    Y coordinate (affine)
```

2.58 LTC Blocking APIs

2.59 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_attrib attribute_memory

struct __mcm_buffer_fault_attrib

#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_attrib attribute_memory

struct __mcm_lmem_fault_attrib

#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.

2.60 MSCM: Miscellaneous System Control

FSL_MSCM_DRIVER_VERSION

MSCM driver version 2.0.0.

typedef struct *_mscm_uid* mscm_uid_t

static inline void MSCM_GetUID(MSCM_Type *base, *mscm_uid_t* *uid)

Get MSCM UID.

Parameters

- base – MSCM peripheral base address.
- uid – Pointer to an uid struct.

static inline void MSCM_SetSecureIrqParameter(MSCM_Type *base, const uint32_t parameter)

Set MSCM Secure Irq.

Parameters

- base – MSCM peripheral base address.
- parameter – Value to be write to SECURE_IRQ.

static inline uint32_t MSCM_GetSecureIrq(MSCM_Type *base)

Get MSCM Secure Irq.

Parameters

- base – MSCM peripheral base address.

Returns

MSCM Secure Irq.

FSL_COMPONENT_ID

struct *_mscm_uid*

#include <fsl_mscm.h>

2.61 MU: Messaging Unit

`uint32_t MU_GetInstance(MU_Type *base)`

Get the MU instance index.

Parameters

- `base` – MU peripheral base address.

Returns

MU instance index.

`void MU_Init(MU_Type *base)`

Initializes the MU module.

This function enables the MU clock only.

Parameters

- `base` – MU peripheral base address.

`void MU_Deinit(MU_Type *base)`

De-initializes the MU module.

This function disables the MU clock only.

Parameters

- `base` – MU peripheral base address.

`static inline void MU_SendMsgNonBlocking(MU_Type *base, uint32_t regIndex, uint32_t msg)`

Writes a message to the TX register.

This function writes a message to the specific TX register. It does not check whether the TX register is empty or not. The upper layer should make sure the TX register is empty before calling this function. This function can be used in ISR for better performance.

```
while (!(kMU_Tx0EmptyFlag & MU_GetStatusFlags(base))) { } Wait for TX0 register empty.
MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG_VAL); Write message to the TX0 register.
```

Parameters

- `base` – MU peripheral base address.
- `regIndex` – TX register index, see `mu_msg_reg_index_t`.
- `msg` – Message to send.

`status_t MU_SendMsg(MU_Type *base, uint32_t regIndex, uint32_t msg)`

Blocks to send a message.

This function waits until the TX register is empty and sends the message. If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and returns `kStatus_Timeout`.

Parameters

- `base` – MU peripheral base address.
- `regIndex` – MU message register, see `mu_msg_reg_index_t`.
- `msg` – Message to send.

Return values

- `kStatus_Success` – Message sent successfully.
- `kStatus_Timeout` – Timeout occurred while waiting for TX register to be empty.

Returns

status_t

static inline uint32_t MU_ReceiveMsgNonBlocking(MU_Type *base, uint32_t regIndex)

Reads a message from the RX register.

This function reads a message from the specific RX register. It does not check whether the RX register is full or not. The upper layer should make sure the RX register is full before calling this function. This function can be used in ISR for better performance.

```
uint32_t msg;
while (!(kMU_Rx0FullFlag & MU_GetStatusFlags(base)))
{
    } Wait for the RX0 register full.

msg = MU_ReceiveMsgNonBlocking(base, kMU_MsgReg0); Read message from RX0 register.
```

Parameters

- base – MU peripheral base address.
- regIndex – RX register index, see mu_msg_reg_index_t.

Returns

The received message.

status_t MU_ReceiveMsgTimeout(MU_Type *base, uint32_t regIndex, uint32_t *readValue)

Blocks to receive a message with timeout protection.

This function waits until the RX register is full and receives the message. If MU1_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

This function provides the same blocking behavior as MU_ReceiveMsg() but with additional timeout protection to prevent system hangs if the other core becomes unresponsive or if hardware issues occur.

Note: Both MU_ReceiveMsg() and MU_ReceiveMsgTimeout() are blocking functions. The difference is that this function includes timeout protection while MU_ReceiveMsg() waits indefinitely.

Parameters

- base – MU peripheral base address.
- regIndex – RX register index, see mu_msg_reg_index_t.
- readValue – Pointer to store the received message.

Return values

- kStatus_Success – Message received successfully.
- kStatus_InvalidArgument – Invalid readValue pointer.
- kStatus_Timeout – Timeout occurred while waiting for RX register to be full.

Returns

status_t

uint32_t MU_ReceiveMsg(MU_Type *base, uint32_t regIndex)

Blocks to receive a message (infinite wait, no timeout protection).

This function waits until the RX register is full and receives the message. This function will wait indefinitely until a message is received.

Note: Both `MU_ReceiveMsg()` and `MU_ReceiveMsgTimeout()` are blocking functions. The difference is that `MU_ReceiveMsgTimeout()` includes timeout protection while this function waits indefinitely.

Warning: This function does not include timeout protection and may cause system hangs if the other core becomes unresponsive. For applications requiring timeout protection, use `MU_ReceiveMsgTimeout()` instead.

Parameters

- `base` – MU peripheral base address.
- `regIndex` – RX register index, see `mu_msg_reg_index_t`.

Returns

The received message.

`static inline void MU_SetFlagsNonBlocking(MU_Type *base, uint32_t flags)`

Sets the 3-bit MU flags reflect on the other MU side.

This function sets the 3-bit MU flags directly. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. The upper layer should make sure the status flag `kMU_FlagsUpdatingFlag` is cleared before calling this function.

```
while (kMU_FlagsUpdatingFlag & MU_GetStatusFlags(base))
{
    } Wait for previous MU flags updating.
MU_SetFlagsNonBlocking(base, 0U); Set the mU flags.
```

Parameters

- `base` – MU peripheral base address.
- `flags` – The 3-bit MU flags to set.

`status_t MU_SetFlags(MU_Type *base, uint32_t flags)`

brief Blocks setting the 3-bit MU flags reflect on the other MU side.

This function blocks setting the 3-bit MU flags. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. This function waits for the MU status flag `kMU_FlagsUpdatingFlag` cleared and sets the 3-bit MU flags.

If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout`.

return `status_t` `retval` `kStatus_Success` Flags were set successfully. `retval` `kStatus_Timeout` Timeout occurred while waiting for flags to update.

Parameters

- base – MU peripheral base address.
- flags – The 3-bit MU flags to set.

static inline uint32_t MU_GetFlags(MU_Type *base)

Gets the current value of the 3-bit MU flags set by the other side.

This function gets the current 3-bit MU flags on the current side.

Parameters

- base – MU peripheral base address.

Returns

flags Current value of the 3-bit flags.

uint32_t MU_GetStatusFlags(MU_Type *base)

Gets the MU status flags.

This function returns the bit mask of the MU status flags. See `_mu_status_flags`.

```
uint32_t flags;
flags = MU_GetStatusFlags(base); Get all status flags.
if (kMU_Tx0EmptyFlag & flags)
{
    The TX0 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG0_VAL);
}
if (kMU_Tx1EmptyFlag & flags)
{
    The TX1 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg1, MSG1_VAL);
}
```

If there are more than 4 general purpose interrupts, use `MU_GetGeneralPurposeStatusFlags`.

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU status flags, see `_mu_status_flags`.

static inline uint32_t MU_GetInterruptsPending(MU_Type *base)

Gets the MU IRQ pending status of enabled interrupts.

This function returns the bit mask of the pending MU IRQs of enabled interrupts. Only these flags are checked. `kMU_Tx0EmptyFlag` `kMU_Tx1EmptyFlag` `kMU_Tx2EmptyFlag` `kMU_Tx3EmptyFlag` `kMU_Rx0FullFlag` `kMU_Rx1FullFlag` `kMU_Rx2FullFlag` `kMU_Rx3FullFlag` `kMU_GenInt0Flag` `kMU_GenInt1Flag` `kMU_GenInt2Flag` `kMU_GenInt3Flag`

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU IRQs pending.

static inline void MU_ClearStatusFlags(MU_Type *base, uint32_t flags)

Clears the specific MU status flags.

This function clears the specific MU status flags. The flags to clear should be passed in as bit mask. See `_mu_status_flags`.

```
Clear general interrupt 0 and general interrupt 1 pending flags.
MU_ClearStatusFlags(base, kMU_GenInt0Flag | kMU_GenInt1Flag);
```

If there are more than 4 general purpose interrupts, use `MU_ClearGeneralPurposeStatusFlags`.

Parameters

- `base` – MU peripheral base address.
- `flags` – Bit mask of the MU status flags. See `_mu_status_flags`. Only the following flags can be cleared by software (if applicable for particular device), other flags are cleared by hardware:
 - `kMU_GenInt0Flag`
 - `kMU_GenInt1Flag`
 - `kMU_GenInt2Flag`
 - `kMU_GenInt3Flag`
 - `kMU_MuResetInterruptFlag`
 - `kMU_OtherSideEnterRunInterruptFlag`
 - `kMU_OtherSideEnterHaltInterruptFlag`
 - `kMU_OtherSideEnterWaitInterruptFlag`
 - `kMU_OtherSideEnterStopInterruptFlag`
 - `kMU_OtherSideEnterPowerDownInterruptFlag`
 - `kMU_ResetAssertInterruptFlag`
 - `kMU_HardwareResetInterruptFlag`

static inline void `MU_EnableInterrupts(MU_Type *base, uint32_t interrupts)`

Enables the specific MU interrupts.

This function enables the specific MU interrupts. The interrupts to enable should be passed in as bit mask. See `_mu_interrupt_enable`.

Enable general interrupt 0 and TX0 empty interrupt.
`MU_EnableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

If there are more than 4 general purpose interrupts, use `MU_EnableGeneralPurposeInterrupts`.

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

static inline void `MU_DisableInterrupts(MU_Type *base, uint32_t interrupts)`

Disables the specific MU interrupts.

This function disables the specific MU interrupts. The interrupts to disable should be passed in as bit mask. See `_mu_interrupt_enable`.

Disable general interrupt 0 and TX0 empty interrupt.
`MU_DisableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

If there are more than 4 general purpose interrupts, use `MU_DisableGeneralPurposeInterrupts`.

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

status_t MU_TriggerInterrupts(MU_Type *base, uint32_t interrupts)

Triggers interrupts to the other core.

This function triggers the specific interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `_mu_interrupt_trigger`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
if (kStatus_Success != MU_TriggerInterrupts(base, kMU_GenInt0InterruptTrigger | kMU_
↪ GenInt2InterruptTrigger))
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

If there are more than 4 general purpose interrupts, use `MU_TriggerGeneralPurposeInterrupts`.

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the interrupts to trigger. See `_mu_interrupt_trigger`.

Return values

- kStatus_Success – Interrupts have been triggered successfully.
- kStatus_Fail – Previous interrupts have not been accepted.

static inline void MU_EnableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)

Enables the MU general purpose interrupts.

This function enables the MU general purpose interrupts. The interrupts to enable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_EnableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to enable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↪ GeneralPurposeInterrupt3);
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the MU general purpose interrupts, see `mu_general_purpose_interrupt_t`.

static inline void MU_DisableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)

Disables the MU general purpose interrupts.

This function disables the MU general purpose interrupts. The interrupts to disable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_DisableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to disable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↪ GeneralPurposeInterrupt3);
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the MU general purpose interrupts. see `mu_general_purpose_interrupt_t`.

static inline uint32_t MU_GetGeneralPurposeStatusFlags(MU_Type *base)

Gets the MU general purpose interrupt status flags.

This function returns the bit mask of the MU general purpose interrupt status flags. `MU_GetStatusFlags` can only get general purpose interrupt status 0~3, this function can get all general purpose interrupts status.

This example shows to check whether general purpose interrupt 0 and 3 happened.

```
uint32_t flags;
flags = MU_GetGeneralPurposeStatusFlags(base);
if (kMU_GeneralPurposeInterrupt0 & flags)
{
}
if (kMU_GeneralPurposeInterrupt3 & flags)
{
}
```

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU general purpose interrupt status flags.

static inline void MU_ClearGeneralPurposeStatusFlags(MU_Type *base, uint32_t flags)

Clear the MU general purpose interrupt status flags.

This function clears the specific MU general purpose interrupt status flags. The flags to clear should be passed in as bit mask. `mu_general_purpose_interrupt_t_mu_status_flags`.

Example to clear general purpose interrupt 0 and general interrupt 1 pending flags.

```
MU_ClearGeneralPurposeStatusFlags(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt1);
```

Parameters

- base – MU peripheral base address.
- flags – Bit mask of the MU general purpose interrupt status flags. See `mu_general_purpose_interrupt_t`.

static inline uint32_t MU_GetRxStatusFlags(MU_Type *base)

Return the RX status flags in reverse numerical order.

This function return the RX status flags in reverse order. Note: RFn bits of SR[3-0](mu status register) are mapped in ascending numerical order: RF0 -> SR[0] RF1 -> SR[1] RF2 -> SR[2] RF3 -> SR[3] This function will return these bits in reverse numerical order(RF3->RF1) to comply with `MU_GetRxStatusFlags()` of mu driver. See `MU_GetRxStatusFlags()` from `drivers/mu/fsl_mu.h`

```
status_reg = MU_GetRxStatusFlags(base);
```

Parameters

- base – MU peripheral base address.

Returns

MU RX status flags in reverse order

status_t MU_TriggerGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)

Triggers general purpose interrupts to the other core.

This function triggers the specific general purpose interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `mu_general_purpose_interrupt_t`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
status_t status;
status = MU_TriggerGeneralPurposeInterrupts(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt2);

if (kStatus_Success != status)
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the interrupts to trigger. See `mu_general_purpose_interrupt_t`.

Return values

- kStatus_Success – Interrupts have been triggered successfully.
- kStatus_Fail – Previous interrupts have not been accepted.

void MU_BootOtherCore(MU_Type *base, mu_core_boot_mode_t mode)

Boots the other core.

This function boots the other core with a boot configuration.

Parameters

- base – MU peripheral base address.
- mode – The other core boot mode.

void MU_HoldOtherCoreReset(MU_Type *base)

Holds the other core reset.

This function causes the other core to be held in reset following any reset event.

Parameters

- base – MU peripheral base address.

static inline *status_t* MU_ResetBothSides(MU_Type *base)

Resets the MU for both A side and B side.

This function resets the MU for both A side and B side. Before reset, it is recommended to interrupt processor B, because this function may affect the ongoing processor B programs.

If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations if waiting for the other side to come out of reset takes too long.

Note: For some platforms, only MU side A could use this function, check reference manual for details.

Parameters

- base – MU peripheral base address.

Return values

- kStatus_Success – The MU was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other side to come out of reset.

Returns

status_t

status_t MU_HardwareResetOtherCore(MU_Type *base, bool waitReset, bool holdReset, mu_core_boot_mode_t bootMode)

Hardware reset the other core.

This function resets the other core, the other core could mask the hardware reset by calling MU_MaskHardwareReset. The hardware reset mask feature is only available for some platforms. This function could be used together with MU_BootOtherCore to control the other core reset workflow.

If MU1_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout if waiting for the other core to enter or exit reset takes too long.

Example 1: Reset the other core, and no hold reset

```
MU_HardwareResetOtherCore(MU_A, true, false, bootMode);
```

In this example, the core at MU side B will reset with the specified boot mode.

Example 2: Reset the other core and hold it, then boot the other core later. Here the other core enters reset, and the reset is hold

```
MU_HardwareResetOtherCore(MU_A, true, true, modeDontCare);
```

Current core boot the other core when necessary.

```
MU_BootOtherCore(MU_A, bootMode);
```

Note: The feature waitReset, holdReset, and bootMode might be not supported for some platforms. waitReset is only available for platforms that FSL_FEATURE_MU_NO_CORE_STATUS not defined as 1 and FSL_FEATURE_MU_HAS_RESET_ASSERT_INT not defined as 0. holdReset is only available for platforms that FSL_FEATURE_MU_HAS_RSTH not defined as 0. bootMode is only available for platforms that FSL_FEATURE_MU_HAS_BOOT not defined as 0.

Parameters

- base – MU peripheral base address.
- waitReset – Wait the other core enters reset. Only work when there is CSSR0[RAIP].
 - true: Wait until the other core enters reset, if the other core has masked the hardware reset, then this function will be blocked.
 - false: Don't wait the reset.
- holdReset – Hold the other core reset or not. Only work when there is CCR0[RSTH].

- true: Hold the other core in reset, this function returns directly when the other core enters reset.
- false: Don't hold the other core in reset, this function waits until the other core out of reset.
- bootMode – Boot mode of the other core, if holdReset is true, this parameter is useless.

Return values

- kStatus_Success – The other core was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other core to enter or exit reset.

Returns

status_t

FSL_MU_DRIVER_VERSION

MU driver version.

enum _mu_status_flags

MU status flags.

Values:

enumerator kMU_Tx0EmptyFlag
TX0 empty.

enumerator kMU_Tx1EmptyFlag
TX1 empty.

enumerator kMU_Tx2EmptyFlag
TX2 empty.

enumerator kMU_Tx3EmptyFlag
TX3 empty.

enumerator kMU_Rx0FullFlag
RX0 full.

enumerator kMU_Rx1FullFlag
RX1 full.

enumerator kMU_Rx2FullFlag
RX2 full.

enumerator kMU_Rx3FullFlag
RX3 full.

enumerator kMU_GenInt0Flag
General purpose interrupt 0 pending.

enumerator kMU_GenInt1Flag
General purpose interrupt 1 pending.

enumerator kMU_GenInt2Flag
General purpose interrupt 2 pending.

enumerator kMU_GenInt3Flag
General purpose interrupt 3 pending.

enumerator kMU_RxFullPendingFlag
Any RX full flag is pending.

enumerator kMU__TxEmptyPendingFlag

Any TX empty flag is pending.

enumerator kMU__GenIntPendingFlag

Any general interrupt flag is pending.

enumerator kMU__EventPendingFlag

MU event pending.

enumerator kMU__FlagsUpdatingFlag

MU flags update is on-going.

enumerator kMU__MuInResetFlag

MU of any side is in reset.

enumerator kMU__MuResetInterruptFlag

The other side initializes MU reset.

enum __mu_interrupt_enable

MU interrupt source to enable.

Values:

enumerator kMU__Tx0EmptyInterruptEnable

TX0 empty.

enumerator kMU__Tx1EmptyInterruptEnable

TX1 empty.

enumerator kMU__Tx2EmptyInterruptEnable

TX2 empty.

enumerator kMU__Tx3EmptyInterruptEnable

TX3 empty.

enumerator kMU__Rx0FullInterruptEnable

RX0 full.

enumerator kMU__Rx1FullInterruptEnable

RX1 full.

enumerator kMU__Rx2FullInterruptEnable

RX2 full.

enumerator kMU__Rx3FullInterruptEnable

RX3 full.

enumerator kMU__GenInt0InterruptEnable

General purpose interrupt 0.

enumerator kMU__GenInt1InterruptEnable

General purpose interrupt 1.

enumerator kMU__GenInt2InterruptEnable

General purpose interrupt 2.

enumerator kMU__GenInt3InterruptEnable

General purpose interrupt 3.

enumerator kMU__MuResetInterruptEnable

The other side initializes MU reset.

enum `_mu_interrupt_trigger`

MU interrupt that could be triggered to the other core.

Values:

enumerator `kMU_GenInt0InterruptTrigger`

General purpose interrupt 0.

enumerator `kMU_GenInt1InterruptTrigger`

General purpose interrupt 1.

enumerator `kMU_GenInt2InterruptTrigger`

General purpose interrupt 2.

enumerator `kMU_GenInt3InterruptTrigger`

General purpose interrupt 3.

enum `_mu_msg_reg_index`

MU message register index.

Values:

enumerator `kMU_MsgReg0`

Message register 0.

enumerator `kMU_MsgReg1`

Message register 1.

enumerator `kMU_MsgReg2`

Message register 2.

enumerator `kMU_MsgReg3`

Message register 3.

enum `_mu_general_purpose_interrupt`

MU general purpose interrupts.

Values:

enumerator `kMU_GeneralPurposeInterrupt0`

General purpose interrupt 0

enumerator `kMU_GeneralPurposeInterrupt1`

General purpose interrupt 1

enumerator `kMU_GeneralPurposeInterrupt2`

General purpose interrupt 2

enumerator `kMU_GeneralPurposeInterrupt3`

General purpose interrupt 3

typedef enum `_mu_msg_reg_index` `mu_msg_reg_index_t`

MU message register index.

typedef enum `_mu_general_purpose_interrupt` `mu_general_purpose_interrupt_t`

MU general purpose interrupts.

`MU_CORE_INTR(intr)`

`MU_MISC_INTR(intr)`

`MU_TX_INTR(intr)`

`MU_RX_INTR(intr)`

MU_GI_INTR(*intr*)
MU_GET_CORE_INTR(*intrs*)
MU_GET_TX_INTR(*intrs*)
MU_GET_RX_INTR(*intrs*)
MU_GET_GI_INTR(*intrs*)
MU_CORE_FLAG(*flag*)
MU_STAT_FLAG(*flag*)
MU_TX_FLAG(*flag*)
MU_RX_FLAG(*flag*)
MU_GI_FLAG(*flag*)
MU_GET_CORE_FLAG(*flags*)
MU_GET_STAT_FLAG(*flags*)
MU_GET_TX_FLAG(*flags*)
MU_GET_RX_FLAG(*flags*)
MU_GET_GI_FLAG(*flags*)

MU1_BUSY_POLL_COUNT

Maximum polling iterations for MU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the MU code before timing out and returning an error.

It applies to all waiting loops in MU driver, such as waiting for TX register to be empty or waiting for RX register to be full.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if a core becomes unresponsive.

2.62 PORT: Port Control and Interrupts

static inline void PORT_GetVersionInfo(PORT_Type *base, *port_version_info_t* *info)

Get PORT version information.

Parameters

- *base* – PORT peripheral base pointer
- *info* – PORT version information

static inline void PORT_SecletPortVoltageRange(PORT_Type *base, *port_voltage_range_t* range)

Get PORT version information.

Note: : PORTA_CONFIG[RANGE] controls the voltage ranges of Port A, B, and C. Read or write PORTB_CONFIG[RANGE] and PORTC_CONFIG[RANGE] does not take effect.

Parameters

- base – PORT peripheral base pointer
- range – port voltage range

```
static inline void PORT_SetPinConfig(PORT_Type *base, uint32_t pin, const port_pin_config_t *config)
```

Sets the port PCR register.

This is an example to define an input pin or output pin PCR configuration.

```
// Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- config – PORT PCR register configuration structure.

```
static inline void PORT_SetMultiplePinsConfig(PORT_Type *base, uint32_t mask, const
                                             port_pin_config_t *config)
```

Sets the port PCR register for multiple pins.

This is an example to define input pins or output pins PCR configuration.

```
Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp ,
    kPORT_PullEnable,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnlockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- config – PORT PCR register configuration structure.

```
static inline void PORT_SetPinMux(PORT_Type *base, uint32_t pin, port_mux_t mux)
```

Configures the pin muxing.

Note: : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- mux – pin muxing slot selection.
 - kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.
 - kPORT_MuxAsGpio : Set as GPIO.
 - kPORT_MuxAlt2 : chip-specific.
 - kPORT_MuxAlt3 : chip-specific.
 - kPORT_MuxAlt4 : chip-specific.
 - kPORT_MuxAlt5 : chip-specific.
 - kPORT_MuxAlt6 : chip-specific.
 - kPORT_MuxAlt7 : chip-specific.

static inline void PORT__EnablePinsDigitalFilter(PORT_Type *base, uint32_t mask, bool enable)
Enables the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- enable – PORT digital filter configuration.

static inline void PORT__SetDigitalFilterConfig(PORT_Type *base, const
port_digital_filter_config_t *config)
Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- config – PORT digital filter configuration structure.

static inline void PORT__SetPinDriveStrength(PORT_Type *base, uint32_t pin, uint8_t strength)
Configures the port pin drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- strength – PORT pin drive strength
 - kPORT_LowDriveStrength = 0U - Low-drive strength is configured.
 - kPORT_HighDriveStrength = 1U - High-drive strength is configured.

static inline void PORT__EnablePinDoubleDriveStrength(PORT_Type *base, uint32_t pin, bool
enable)

Enables the port pin double drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- enable – PORT pin drive strength configuration.

static inline void PORT_SetPinPullValue(PORT_Type *base, uint32_t pin, uint8_t value)

Configures the port pin pull value.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- value – PORT pin pull value
 - kPORT_LowPullResistor = 0U - Low internal pull resistor value is selected.
 - kPORT_HighPullResistor = 1U - High internal pull resistor value is selected.

static inline uint32_t PORT_GetEFTDetectFlags(PORT_Type *base)

Get EFT detect flags.

Parameters

- base – PORT peripheral base pointer

Returns

EFT detect flags

static inline void PORT_EnableEFTDetectInterrupts(PORT_Type *base, uint32_t interrupt)

Enable EFT detect interrupts.

Parameters

- base – PORT peripheral base pointer
- interrupt – EFT detect interrupt

static inline void PORT_DisableEFTDetectInterrupts(PORT_Type *base, uint32_t interrupt)

Disable EFT detect interrupts.

Parameters

- base – PORT peripheral base pointer
- interrupt – EFT detect interrupt

static inline void PORT_ClearAllLowEFTDetectors(PORT_Type *base)

Clear all low EFT detector.

Note: : Port B and Port C pins share the same EFT detector clear control from PORTC_EDCR register. Any write to the PORTB_EDCR does not take effect.

Parameters

- base – PORT peripheral base pointer

static inline void PORT_ClearAllHighEFTDetectors(PORT_Type *base)

Clear all high EFT detector.

Parameters

- base – PORT peripheral base pointer

FSL_PORT_DRIVER_VERSION

PORT driver version.

enum __port_pull

Internal resistor pull feature selection.

Values:

enumerator kPORT_PullDisable

Internal pull-up/down resistor is disabled.

enumerator kPORT_PullDown

Internal pull-down resistor is enabled.

enumerator kPORT_PullUp

Internal pull-up resistor is enabled.

enum __port_pull_value

Internal resistor pull value selection.

Values:

enumerator kPORT_LowPullResistor

Low internal pull resistor value is selected.

enumerator kPORT_HighPullResistor

High internal pull resistor value is selected.

enum __port_slew_rate

Slew rate selection.

Values:

enumerator kPORT_FastSlewRate

Fast slew rate is configured.

enumerator kPORT_SlowSlewRate

Slow slew rate is configured.

enum __port_open_drain_enable

Open Drain feature enable/disable.

Values:

enumerator kPORT_OpenDrainDisable

Open drain output is disabled.

enumerator kPORT_OpenDrainEnable

Open drain output is enabled.

enum __port_passive_filter_enable

Passive filter feature enable/disable.

Values:

enumerator kPORT_PassiveFilterDisable

Passive input filter is disabled.

enumerator kPORT_PassiveFilterEnable

Passive input filter is enabled.

enum __port_drive_strength

Configures the drive strength.

Values:

enumerator kPORT_LowDriveStrength

Low-drive strength is configured.

enumerator kPORT_HighDriveStrength

High-drive strength is configured.

enum __port_drive_strength1

Configures the drive strength1.

Values:

enumerator kPORT_NormalDriveStrength

Normal drive strength

enumerator kPORT_DoubleDriveStrength

Double drive strength

enum __port_invet_input

Digital input is not inverted or it is inverted.

Values:

enumerator kPORT_InputNormal

Digital input is not inverted

enumerator kPORT_InputInvert

Digital input is inverted

enum __port_lock_register

Unlock/lock the pin control register field[15:0].

Values:

enumerator kPORT_UnlockRegister

Pin Control Register fields [15:0] are not locked.

enumerator kPORT_LockRegister

Pin Control Register fields [15:0] are locked.

enum __port_mux

Pin mux selection.

Values:

enumerator kPORT_PinDisabledOrAnalog

Corresponding pin is disabled, but is used as an analog pin.

enumerator kPORT_MuxAsGpio

Corresponding pin is configured as GPIO.

enumerator kPORT_MuxAlt0

Chip-specific

enumerator kPORT_MuxAlt1

Chip-specific

enumerator kPORT_MuxAlt2

Chip-specific

enumerator kPORT_MuxAlt3

Chip-specific

enumerator kPORT_MuxAlt4

Chip-specific

enumerator kPORT_MuxAlt5

Chip-specific


```

enumerator kPORT_MuxAlt6
    Chip-specific
enumerator kPORT_MuxAlt7
    Chip-specific
enumerator kPORT_MuxAlt8
    Chip-specific
enumerator kPORT_MuxAlt9
    Chip-specific
enumerator kPORT_MuxAlt10
    Chip-specific
enumerator kPORT_MuxAlt11
    Chip-specific
enumerator kPORT_MuxAlt12
    Chip-specific
enumerator kPORT_MuxAlt13
    Chip-specific
enumerator kPORT_MuxAlt14
    Chip-specific
enumerator kPORT_MuxAlt15
    Chip-specific
enum _port_digital_filter_clock_source
    Digital filter clock source selection.
    Values:
    enumerator kPORT_BusClock
        Digital filters are clocked by the bus clock.
    enumerator kPORT_LpoClock
        Digital filters are clocked by the 1 kHz LPO clock.
enum _port_voltage_range
    PORT voltage range.
    Values:
    enumerator kPORT_VoltageRange1Dot71V_3Dot6V
        Port voltage range is 1.71 V - 3.6 V.
    enumerator kPORT_VoltageRange2Dot70V_3Dot6V
        Port voltage range is 2.70 V - 3.6 V.
typedef enum _port_mux port_mux_t
    Pin mux selection.
typedef enum _port_digital_filter_clock_source port_digital_filter_clock_source_t
    Digital filter clock source selection.
typedef struct _port_digital_filter_config port_digital_filter_config_t
    PORT digital filter feature configuration definition.
typedef struct _port_pin_config port_pin_config_t
    PORT pin configuration structure.

```

```
typedef struct _port_version_info port_version_info_t
    PORT version information.
```

```
typedef enum _port_voltage_range port_voltage_range_t
    PORT voltage range.
```

```
FSL_COMPONENT_ID
```

```
struct _port_digital_filter_config
    #include <fsl_port.h> PORT digital filter feature configuration definition.
```

Public Members

```
uint32_t digitalFilterWidth
    Set digital filter width

port_digital_filter_clock_source_t clockSource
    Set digital filter clockSource
```

```
struct _port_pin_config
    #include <fsl_port.h> PORT pin configuration structure.
```

Public Members

```
uint16_t pullSelect
    No-pull/pull-down/pull-up select

uint16_t pullValueSelect
    Pull value select

uint16_t slewRate
    Fast/slow slew rate Configure

uint16_t passiveFilterEnable
    Passive filter enable/disable

uint16_t openDrainEnable
    Open drain enable/disable

uint16_t driveStrength
    Fast/slow drive strength configure

uint16_t driveStrength1
    Normal/Double drive strength enable/disable

uint16_t invertInput
    Invert Input Configure

uint16_t lockRegister
    Lock/unlock the PCR field[15:0]
```

```
struct _port_version_info
    #include <fsl_port.h> PORT version information.
```

Public Members

```
uint16_t feature
    Feature Specification Number.
```

uint8_t minor

Minor Version Number.

uint8_t major

Major Version Number.

2.63 RTC: Real Time Clock

void RTC_Init(RTC_Type *base, const *rtc_config_t* *config)

Ungates the RTC clock and configures the peripheral for basic operation.

This function issues a software reset if the timer invalid flag is set.

Note: This API should be called at the beginning of the application using the RTC driver.

Parameters

- base – RTC peripheral base address
- config – Pointer to the user's RTC configuration structure.

static inline void RTC_Deinit(RTC_Type *base)

Stops the timer and gate the RTC clock.

Parameters

- base – RTC peripheral base address

void RTC_GetDefaultConfig(*rtc_config_t* *config)

Fills in the RTC config struct with the default settings.

The default values are as follows.

```
config->clockOutput = false;
config->wakeupSelect = false;
config->updateMode = false;
config->supervisorAccess = false;
config->compensationInterval = 0;
config->compensationTime = 0;
```

Parameters

- config – Pointer to the user's RTC configuration structure.

status_t RTC_SetDatetime(RTC_Type *base, const *rtc_datetime_t* *datetime)

Sets the RTC date and time according to the given time structure.

The RTC counter must be stopped prior to calling this function because writes to the RTC seconds register fail if the RTC counter is running.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to the structure where the date and time details are stored.

Returns

kStatus_Success: Success in setting the time and starting the RTC
 kStatus_InvalidArgument: Error because the datetime format is incorrect

`void RTC_GetDatetime(RTC_Type *base, rtc_datetime_t *datetime)`

Gets the RTC time and stores it in the given time structure.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to the structure where the date and time details are stored.

`status_t RTC_SetAlarm(RTC_Type *base, const rtc_datetime_t *alarmTime)`

Sets the RTC alarm time.

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

- `base` – RTC peripheral base address
- `alarmTime` – Pointer to the structure where the alarm time is stored.

Returns

`kStatus_Success`: success in setting the RTC alarm
`kStatus_InvalidArgument`: Error because the alarm datetime format is incorrect
`kStatus_Fail`: Error because the alarm time has already passed

`void RTC_GetAlarm(RTC_Type *base, rtc_datetime_t *datetime)`

Returns the RTC alarm time.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to the structure where the alarm date and time details are stored.

`void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)`

Enables the selected RTC interrupts.

Parameters

- `base` – RTC peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `rtc_interrupt_enable_t`

`void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)`

Disables the selected RTC interrupts.

Parameters

- `base` – RTC peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `rtc_interrupt_enable_t`

`uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)`

Gets the enabled RTC interrupts.

Parameters

- `base` – RTC peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `rtc_interrupt_enable_t`

uint32_t RTC_GetStatusFlags(RTC_Type *base)

Gets the RTC status flags.

Parameters

- base – RTC peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `rtc_status_flags_t`

void RTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)

Clears the RTC status flags.

Parameters

- base – RTC peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `rtc_status_flags_t`

static inline void RTC_EnableLPOClock(RTC_Type *base, bool enable)

Enable/Disable RTC 1kHz LPO clock.

Note: After setting this bit, RTC prescaler increments using the LPO 1kHz clock and not the RTC 32kHz crystal clock.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC 1kHz LPO clock

static inline void RTC_StartTimer(RTC_Type *base)

Starts the RTC time counter.

After calling this function, the timer counter increments once a second provided `SR[TOF]` or `SR[TIF]` are not set.

Parameters

- base – RTC peripheral base address

static inline void RTC_StopTimer(RTC_Type *base)

Stops the RTC time counter.

RTC's seconds register can be written to only when the timer is stopped.

Parameters

- base – RTC peripheral base address

void RTC_GetMonotonicCounter(RTC_Type *base, uint64_t *counter)

Reads the values of the Monotonic Counter High and Monotonic Counter Low and returns them as a single value.

Parameters

- base – RTC peripheral base address
- counter – Pointer to variable where the value is stored.

void RTC_SetMonotonicCounter(RTC_Type *base, uint64_t counter)

Writes values Monotonic Counter High and Monotonic Counter Low by decomposing the given single value. The Monotonic Overflow Flag in `RTC_SR` is cleared due to the API.

Parameters

- base – RTC peripheral base address
- counter – Counter value

`status_t RTC_IncrementMonotonicCounter(RTC_Type *base)`

Increments the Monotonic Counter by one.

Increments the Monotonic Counter (registers RTC_MCLR and RTC_MCHR accordingly) by setting the monotonic counter enable (MER[MCE]) and then writing to the RTC_MCLR register. A write to the monotonic counter low that causes it to overflow also increments the monotonic counter high.

Parameters

- base – RTC peripheral base address

Returns

kStatus_Success: success
kStatus_Fail: error occurred, either time invalid or monotonic overflow flag was found

FSL_RTC_DRIVER_VERSION

Version 2.4.0

enum _rtc_interrupt_enable

List of RTC interrupts.

Values:

enumerator kRTC_TimeInvalidInterruptEnable

Time invalid interrupt.

enumerator kRTC_TimeOverflowInterruptEnable

Time overflow interrupt.

enumerator kRTC_AlarmInterruptEnable

Alarm interrupt.

enumerator kRTC_MonotonicOverflowInterruptEnable

Monotonic Overflow Interrupt Enable

enumerator kRTC_SecondsInterruptEnable

Seconds interrupt.

enumerator kRTC_TestModeInterruptEnable

enumerator kRTC_FlashSecurityInterruptEnable

enumerator kRTC_TamperPinInterruptEnable

enumerator kRTC_SecurityModuleInterruptEnable

enumerator kRTC_LossOfClockInterruptEnable

enum _rtc_status_flags

List of RTC flags.

Values:

enumerator kRTC_TimeInvalidFlag

Time invalid flag

enumerator kRTC_TimeOverflowFlag

Time overflow flag

enumerator kRTC_AlarmFlag

Alarm flag

enumerator kRTC_MonotonicOverflowFlag
Monotonic Overflow Flag

enumerator kRTC_TamperInterruptDetectFlag
Tamper interrupt detect flag

enumerator kRTC_TestModeFlag

enumerator kRTC_FlashSecurityFlag

enumerator kRTC_TamperPinFlag

enumerator kRTC_SecurityTamperFlag

enumerator kRTC_LossOfClockTamperFlag

enum _rtc_osc_cap_load
List of RTC Oscillator capacitor load settings.
Values:

enumerator kRTC_Capacitor_2p
2 pF capacitor load

enumerator kRTC_Capacitor_4p
4 pF capacitor load

enumerator kRTC_Capacitor_8p
8 pF capacitor load

enumerator kRTC_Capacitor_16p
16 pF capacitor load

enum _rtc_timer_seconds_interrupt_frequency
List of RTC Timer Seconds Interrupt Frequencies.
Values:

enumerator kRTC_TimerSecondsFrequency1Hz
Timer seconds frequency is 1Hz

enumerator kRTC_TimerSecondsFrequency2Hz
Timer seconds frequency is 2Hz

enumerator kRTC_TimerSecondsFrequency4Hz
Timer seconds frequency is 4Hz

enumerator kRTC_TimerSecondsFrequency8Hz
Timer seconds frequency is 8Hz

enumerator kRTC_TimerSecondsFrequency16Hz
Timer seconds frequency is 16Hz

enumerator kRTC_TimerSecondsFrequency32Hz
Timer seconds frequency is 32Hz

enumerator kRTC_TimerSecondsFrequency64Hz
Timer seconds frequency is 64Hz

enumerator kRTC_TimerSecondsFrequency128Hz
Timer seconds frequency is 128Hz

typedef enum _rtc_interrupt_enable rtc_interrupt_enable_t
List of RTC interrupts.

```
typedef enum _rtc_status_flags rtc_status_flags_t
```

List of RTC flags.

```
typedef enum _rtc_osc_cap_load rtc_osc_cap_load_t
```

List of RTC Oscillator capacitor load settings.

```
typedef enum _rtc_timer_seconds_interrupt_frequency rtc_timer_seconds_interrupt_frequency_t
```

List of RTC Timer Seconds Interrupt Frequencies.

```
typedef struct _rtc_datetime rtc_datetime_t
```

Structure is used to hold the date and time.

```
typedef struct _rtc_pin_config rtc_pin_config_t
```

RTC pin config structure.

```
typedef struct _rtc_config rtc_config_t
```

RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the `RTC_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

```
static inline uint32_t RTC_GetTamperTimeSeconds(RTC_Type *base)
```

Get the RTC tamper time seconds.

Parameters

- `base` – RTC peripheral base address

```
static inline void RTC_SetOscCapLoad(RTC_Type *base, uint32_t capLoad)
```

This function sets the specified capacitor configuration for the RTC oscillator.

Parameters

- `base` – RTC peripheral base address
- `capLoad` – Oscillator loads to enable. This is a logical OR of members of the enumeration `rtc_osc_cap_load_t`

```
static inline void RTC_Reset(RTC_Type *base)
```

Performs a software reset on the RTC module.

This resets all RTC registers except for the SWR bit and the `RTC_WAR` and `RTC_RAR` registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

- `base` – RTC peripheral base address

```
static inline void RTC_EnableWakeUpPin(RTC_Type *base, bool enable)
```

Enables or disables the RTC Wakeup Pin Operation.

This function enable or disable RTC Wakeup Pin. The wakeup pin is optional and not available on all devices.

Parameters

- `base` – `RTC_Type` base pointer.
- `enable` – true to enable, false to disable.

```
static inline void RTC_EnableClockOutput(RTC_Type *base, bool enable)
```

Enables or disables the RTC 32 kHz clock output.

This function enables or disables the RTC 32 kHz clock output.

Parameters

- base – RTC_Type base pointer.
- enable – true to enable, false to disable.

```
void RTC_SetTimerSecondsInterruptFrequency(RTC_Type *base,  
                                           rtc_timer_seconds_interrupt_frequency_t freq)
```

Sets the RTC timer seconds interrupt frequency.

This function sets the RTC timer seconds interrupt frequency.

Parameters

- base – RTC peripheral base address
- freq – The timer seconds interrupt frequency. This is a member of the enumeration `rtc_timer_seconds_interrupt_frequency_t`

```
struct _rtc_datetime
```

#include <fsl_rtc.h> Structure is used to hold the date and time.

Public Members

```
uint16_t year
```

Range from 1970 to 2099.

```
uint8_t month
```

Range from 1 to 12.

```
uint8_t day
```

Range from 1 to 31 (depending on month).

```
uint8_t hour
```

Range from 0 to 23.

```
uint8_t minute
```

Range from 0 to 59.

```
uint8_t second
```

Range from 0 to 59.

```
struct _rtc_pin_config
```

#include <fsl_rtc.h> RTC pin config structure.

Public Members

```
bool inputLogic
```

true: Tamper pin input data is logic one. false: Tamper pin input data is logic zero.

```
bool pinActiveLow
```

true: Tamper pin is active low. false: Tamper pin is active high.

```
bool filterEnable
```

true: Input filter is enabled on the tamper pin. false: Input filter is disabled on the tamper pin.

```
bool pullSelectNegate
```

true: Tamper pin pull resistor direction will negate the tamper pin. false: Tamper pin pull resistor direction will assert the tamper pin.

```
bool pullEnable
```

true: Pull resistor is enabled on tamper pin. false: Pull resistor is disabled on tamper pin.

struct `_rtc_config`

#include <fsl_rtc.h> RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the `RTC_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

bool `clockOutput`

true: The 32 kHz clock is not output to other peripherals; false: The 32 kHz clock is output to other peripherals

bool `wakeupSelect`

true: Wakeup pin outputs the 32 KHz clock; false: Wakeup pin used to wakeup the chip

bool `updateMode`

true: Registers can be written even when locked under certain conditions, false: No writes allowed when registers are locked

bool `supervisorAccess`

true: Non-supervisor accesses are allowed; false: Non-supervisor accesses are not supported

uint32_t `compensationInterval`

Compensation interval that is written to the CIR field in RTC TCR Register

uint32_t `compensationTime`

Compensation time that is written to the TCR field in RTC TCR Register

2.64 SEMA42: Hardware Semaphores Driver

FSL_SEMA42_DRIVER_VERSION

SEMA42 driver version.

SEMA42 status return codes.

Values:

enumerator `kStatus_SEMA42_Busy`

SEMA42 gate has been locked by other processor.

enumerator `kStatus_SEMA42_Reseting`

SEMA42 gate reseting is ongoing.

enum `_sema42_gate_status`

SEMA42 gate lock status.

Values:

enumerator `kSEMA42_Unlocked`

The gate is unlocked.

enumerator `kSEMA42_LockedByProc0`

The gate is locked by processor 0.

enumerator kSEMA42_LockedByProc1

The gate is locked by processor 1.

enumerator kSEMA42_LockedByProc2

The gate is locked by processor 2.

enumerator kSEMA42_LockedByProc3

The gate is locked by processor 3.

enumerator kSEMA42_LockedByProc4

The gate is locked by processor 4.

enumerator kSEMA42_LockedByProc5

The gate is locked by processor 5.

enumerator kSEMA42_LockedByProc6

The gate is locked by processor 6.

enumerator kSEMA42_LockedByProc7

The gate is locked by processor 7.

enumerator kSEMA42_LockedByProc8

The gate is locked by processor 8.

enumerator kSEMA42_LockedByProc9

The gate is locked by processor 9.

enumerator kSEMA42_LockedByProc10

The gate is locked by processor 10.

enumerator kSEMA42_LockedByProc11

The gate is locked by processor 11.

enumerator kSEMA42_LockedByProc12

The gate is locked by processor 12.

enumerator kSEMA42_LockedByProc13

The gate is locked by processor 13.

enumerator kSEMA42_LockedByProc14

The gate is locked by processor 14.

typedef enum *_sema42_gate_status* sema42_gate_status_t
SEMA42 gate lock status.

void SEMA42_Init(SEMA42_Type *base)

Initializes the SEMA42 module.

This function initializes the SEMA42 module. It only enables the clock but does not reset the gates because the module might be used by other processors at the same time. To reset the gates, call either SEMA42_ResetGate or SEMA42_ResetAllGates function.

Parameters

- base – SEMA42 peripheral base address.

void SEMA42_Deinit(SEMA42_Type *base)

De-initializes the SEMA42 module.

This function de-initializes the SEMA42 module. It only disables the clock.

Parameters

- base – SEMA42 peripheral base address.

status_t SEMA42_TryLock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Tries to lock the SEMA42 gate.

This function tries to lock the specific SEMA42 gate. If the gate has been locked by another processor, this function returns an error code.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – Lock the sema42 gate successfully.
- kStatus_SEMA42_Busy – Sema42 gate has been locked by another processor.

status_t SEMA42_Lock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Locks the SEMA42 gate.

This function locks the specific SEMA42 gate. If the gate has been locked by other processors, this function waits until it is unlocked and then lock it.

If SEMA42_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – The gate was successfully locked.
- kStatus_Timeout – Timeout occurred while waiting for the gate to be unlocked.

Returns

status_t

static inline void SEMA42_Unlock(SEMA42_Type *base, uint8_t gateNum)

Unlocks the SEMA42 gate.

This function unlocks the specific SEMA42 gate. It only writes unlock value to the SEMA42 gate register. However, it does not check whether the SEMA42 gate is locked by the current processor or not. As a result, if the SEMA42 gate is not locked by the current processor, this function has no effect.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to unlock.

static inline *sema42_gate_status_t* SEMA42_GetGateStatus(SEMA42_Type *base, uint8_t gateNum)

Gets the status of the SEMA42 gate.

This function checks the lock status of a specific SEMA42 gate.

Parameters

- base – SEMA42 peripheral base address.

- gateNum – Gate number.

Returns

status Current status.

status_t SEMA42_ResetGate(SEMA42_Type *base, uint8_t gateNum)

Resets the SEMA42 gate to an unlocked status.

This function resets a SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number.

Return values

- kStatus_Success – SEMA42 gate is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

static inline *status_t* SEMA42_ResetAllGates(SEMA42_Type *base)

Resets all SEMA42 gates to an unlocked status.

This function resets all SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.

Return values

- kStatus_Success – SEMA42 is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

SEMA42_GATE_NUM_RESET_ALL

The number to reset all SEMA42 gates.

SEMA42_GATEn(base, n)

SEMA42 gate n register address.

The SEMA42 gates are sorted in the order 3, 2, 1, 0, 7, 6, 5, 4, ... not in the order 0, 1, 2, 3, 4, 5, 6, 7, ... The macro SEMA42_GATEn gets the SEMA42 gate based on the gate index.

The input gate index is XOR'ed with 3U: $0 \wedge 3 = 3$ $1 \wedge 3 = 2$ $2 \wedge 3 = 1$ $3 \wedge 3 = 0$ $4 \wedge 3 = 7$ $5 \wedge 3 = 6$ $6 \wedge 3 = 5$ $7 \wedge 3 = 4$...

SEMA42_BUSY_POLL_COUNT

Maximum polling iterations for SEMA42 waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the SEMA42 driver code before timing out and returning an error.

It applies to all waiting loops in SEMA42 driver, such as waiting for a gate to be unlocked, waiting for a reset to complete, or waiting for a resource to become available.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if hardware doesn't respond or if a resource is never released.

2.65 SFA: Signal Frequency Analyser

`void SFA_GetDefaultConfig(sfa_config_t *config)`

Fill the SFA configuration structure with default settings.

The default values are:

```
config->mode = kSFA_FrequencyMeasurement0;
config->cutSelect = kSFA_CUTSelect0;
config->refSelect = kSFA_REFSelect0;
config->prediv = 0U;
config->trigStart = kSFA_TriggerStartSelect0;
config->startPolarity = kSFA_TriggerStartPolarityRiseEdge;
config->trigEnd = kSFA_TriggerEndSelect0;
config->endPolarity = kSFA_TriggerEndPolarityRiseEdge;
config->enableTrigMeasurement = false;
config->enableCUTPin = false;
config->cutTarget = 0xffffU;
config->refTarget = 0xffffffffU;
```

Parameters

- `config` – Pointer to the user configuration structure.

`void SFA_Init(SFA_Type *base)`

Initialize SFA.

Parameters

- `base` – SFA peripheral base address.

`status_t SFA_Deinit(SFA_Type *base)`

Clear counter, disable SFA and gate the SFA clock.

Parameters

- `base` – SFA peripheral base address.

Return values

- `kStatus_Success` – run success.
- `kStatus_Timeout` – timeout occurs.

`static inline void SFA_EnableCUTPin(SFA_Type *base, bool enable)`

Control the connection of the clock under test to an external pin.

Parameters

- `base` – SFA peripheral base address.
- `enable` – `true`: connect the clock under test and external pin. `false`: Disconnect the clock under test and external pin.

`static inline uint32_t SFA_GetStatusFlags(SFA_Type *base)`

Get SFA status flags.

Parameters

- `base` – SFA peripheral base address.

`void SFA_ClearStatusFlag(SFA_Type *base, uint32_t mask)`

Clear the SFA status flags.

Note: To clear `kSFA_RefStoppedFlag`, `kSFA_CUTStoppedFlag`, `kSFA_MeasurementStartedFlag`, and `kSFA_ReferenceCounterTimeOutFlag`, each counter will also be cleared.

Parameters

- base – SFA peripheral base address.
- mask – SFA status flag mask (see `_sfa_status_flags` for bit definition).

```
static inline void SFA__EnableInterrupts(SFA_Type *base, uint32_t mask)
```

Enable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to enable (see `_sfa_interrupts_enable` for definition).

```
static inline void SFA__DisableInterrupts(SFA_Type *base, uint32_t mask)
```

Disable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to disable (see `_sfa_interrupts_enable` for definition).

```
static inline uint8_t SFA__GetMode(SFA_Type *base)
```

Get SFA measurement mode.

Parameters

- base – SFA peripheral base address.

```
static inline uint8_t SFA__GetCUTPredivide(SFA_Type *base)
```

Get CUT predivide value.

Parameters

- base – SFA peripheral base address.

```
void SFA__InstallCallback(SFA_Type *base, sfa_callback_t function)
```

Install the callback function to be called when IRQ happens or measurement completes.

Parameters

- base – SFA peripheral base address.
- function – the SFA measure completed callback function.

```
void SFA__SetMeasureConfig(SFA_Type *base, const sfa_config_t *config)
```

Set Measurement options with the passed in configuration structure.

Parameters

- base – SFA peripheral base address.
- config – SFA configuration structure.

```
status_t SFA__MeasureBlocking(SFA_Type *base)
```

Start SFA measurement in blocking mode.

Parameters

- base – SFA peripheral base address.

Return values

- `kStatus_SFA_MeasurementCompleted` – SFA measure completes.
- `kStatus_SFA_ReferenceCounterTimeout` – reference counter timeout error happens.
- `kStatus_SFA_CUTCounterTimeout` – CUT counter time out happens.

void SFA_MeasureNonBlocking(SFA_Type *base)

Start measure sequence in NonBlocking mode.

This function performs nonblocking measurement by enabling sfa interrupt (Please enable the FreqGreaterThanMax and FreqLessThanMin interrupts individually as needed). The callback function must be installed before invoking this function.

Note: This function has different functions for different instances.

Parameters

- base – SFA peripheral base address.

void SFA_AbortMeasureSequence(SFA_Type *base)

Abort SFA measurement sequence.

Parameters

- base – SFA peripheral base address.

uint32_t SFA_CalculateFrequencyOrPeriod(SFA_Type *base, uint32_t refFrequency)

Calculate the frequency or period.

Parameters

- base – SFA peripheral base address.
- refFrequency – The reference clock frequency(BUS clock recommended).

static inline uint32_t SFA_GetCUTCOUNTER(SFA_Type *base)

Get current count of the clock under test.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTTargetCount(SFA_Type *base, uint32_t count)

Set the target count for the clock under test.

Parameters

- base – SFA peripheral base address.
- count – target count for CUT.

static inline uint32_t SFA_GetCUTTargetCount(SFA_Type *base)

Get the target count of the clock under test.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTLowLimitClockCount(SFA_Type *base, uint32_t count)

Set CUT low limit clock count.

Parameters

- base – SFA peripheral base address.
- count – low limit count for CUT clock.

static inline uint32_t SFA_GetCUTLowLimitClockCount(SFA_Type *base)

Get CUT low limit clock count.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTHighLimitClockCount(SFA_Type *base, uint32_t count)
Set CUT high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for CUT clock.

static inline uint32_t SFA_GetCUTHighLimitClockCount(SFA_Type *base)
Get CUT high limit clock count.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFCounter(SFA_Type *base)
Get current count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFTargetCount(SFA_Type *base, uint32_t count)
Set the target count for the reference clock.

Parameters

- base – SFA peripheral base address.
- count – target count for reference clock.

static inline uint32_t SFA_GetREFTargetCount(SFA_Type *base)
Get the target count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFStartCount(SFA_Type *base)
Get saved reference clock counter which is loaded when measurement start.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFEndCount(SFA_Type *base)
Get saved reference clock counter which is loaded when measurement complete.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFLowLimitClockCount(SFA_Type *base, uint32_t count)
Set REF low limit clock count.

Parameters

- base – SFA peripheral base address.
- count – low limit count for REF clock.

static inline uint32_t SFA_GetREFLowLimitClockCount(SFA_Type *base)
Get REF low limit clock count.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFHighLimitClockCount(SFA_Type *base, uint32_t count)

Set REF high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for REF clock.

static inline uint32_t SFA_GetREFHighLimitClockCount(SFA_Type *base)

Get REF high limit clock count.

Parameters

- base – SFA peripheral base address.

FSL_SFA_DRIVER_VERSION

SFA driver version 2.1.4.

SFA_MEASUREMENT_START_TIMEOUT

Max loops to wait for SFA measurement started.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_CUT_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA CUT counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_REF_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA REF counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA status return codes.

enumeration _sfa_status

Values:

enumerator kStatus_SFA_MeasurementCompleted

Measurement completed

enumerator kStatus_SFA_ReferenceCounterTimeout

Reference counter timeout

enumerator kStatus_SFA_CUTCounterTimeout

CUT counter timeout

enumerator kStatus_SFA_CUTClockFreqLessThanMinLimit

CUT clock frequency less than minimum limit

enumerator kStatus_SFA_CUTClockFreqGreaterThanMaxLimit

CUT clock frequency greater than maximum limit

enum _sfa_status_flags

List of SFA status flags.

The following status register flags can be cleared on any write to REF_CNT.

- kSFA_RefStoppedFlag
- kSFA_CutStoppedFlag

- kSFA_MeasurementStartedFlag
- kSFA_ReferenceCounterTimeOutFlag

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kSFA_RefStoppedFlag

Reference counter stopped flag

enumerator kSFA_CutStoppedFlag

CUT counter stopped flag

enumerator kSFA_MeasurementStartedFlag

Measurement Started flag

enumerator kSFA_ReferenceCounterTimeOutFlag

Reference counter time out flag

enumerator kSFA_InterruptRequestFlag

SFA interrupt request flag

enumerator kSFA_FreqGreaterThanMaxInterruptFlag

FREQ_GT_MAX interrupt flag

enumerator kSFA_FreqLessThanMinInterruptFlag

FREQ_LT_MIN interrupt flag

enumerator kSFA_AllStatusFlags

enum _sfa_interrupts_enable

List of SFA interrupt.

Values:

enumerator kSFA_InterruptEnable

SFA interrupt enable

enumerator kSFA_FreqGreaterThanMaxInterruptEnable

FREQ_GT_MAX interrupt enable

enumerator kSFA_FreqLessThanMinInterruptEnable

FREQ_LT_MIN interrupt enable

enum _sfa_measurement_mode

List of SFA measurement mode(Please check the mode configuration according to the manual).

Values:

enumerator kSFA_FrequencyMeasurement0

Frequency measurement performed with REF frequency > CUT frequency

enumerator kSFA_FrequencyMeasurement1

Frequency measurement performed with REF frequency < CUT frequency

enumerator kSFA_CUTPeriodMeasurement

CUT period measurement performed

enumerator kSFA_TriggerBasedMeasurement

Trigger based measurement performed

enum _sfa_cut_select

List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

Values:

enumerator kSFA__CUTSelect0

enumerator kSFA__CUTSelect1

enumerator kSFA__CUTSelect2

enumerator kSFA__CUTSelect3

enumerator kSFA__CUTSelect4

enumerator kSFA__CUTSelect5

enumerator kSFA__CUTSelect6

enumerator kSFA__CUTSelect7

enumerator kSFA__CUTSelect8

enumerator kSFA__CUTSelect9

enumerator kSFA__CUTSelect10

enumerator kSFA__CUTSelect11

enumerator kSFA__CUTSelect12

enumerator kSFA__CUTSelect13

enumerator kSFA__CUTSelect14

enumerator kSFA__CUTSelect15

enum _sfa_ref_select

List of REF which is connected to the REF counter (Please refer to the manual for configuration).

Values:

enumerator kSFA__REFSelect0

enumerator kSFA__REFSelect1

enumerator kSFA__REFSelect2

enum _sfa_trigger_start_select

List of Signal MUX for Trigger Based Measurement Start.

Values:

enumerator kSFA__TriggerStartSelect0

enumerator kSFA__TriggerStartSelect1

enum _sfa_trigger_end_select

List of Signal MUX for Trigger Based Measurement End.

Values:

enumerator kSFA__TriggerEndSelect0

enumerator kSFA_TriggerEndSelect1

enum _sfa_trigger_start_polarity

List of Trigger Start Polarity.

Values:

enumerator kSFA_TriggerStartPolarityRiseEdge

Rising edge will begin the measurement sequence

enumerator kSFA_TriggerStartPolarityFallEdge

Falling edge will begin the measurement sequence

enum _sfa_trigger_end_polarity

List of Trigger End Polarity.

Values:

enumerator kSFA_TriggerEndPolarityRiseEdge

Rising edge will end the measurement sequence

enumerator kSFA_TriggerEndPolarityFallEdge

Falling edge will end the measurement sequence

typedef void (*sfa_callback_t)(status_t status)

sfa measure completion callback function pointer type

This callback can be used in non blocking IRQHandler. Specify the callback you want in the call to SFA_InstallCallback().

Param status

The runtime measurement status. kStatus_SFA_MeasurementCompleted: The measurement completes. kStatus_SFA_ReferenceCounterTimeout: Reference counter timeout happens. kStatus_SFA_CUTCounterTimeout: CUT counter timeout happens.

typedef enum _sfa_measurement_mode sfa_measurement_mode_t

List of SFA measurement mode(Please check the mode configuration according to the manual).

typedef enum _sfa_cut_select sfa_cut_select_t

List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

typedef enum _sfa_ref_select sfa_ref_select_t

List of REF which is connected to the REF counter (Please refer to the manual for configuration).

typedef enum _sfa_trigger_start_select sfa_trigger_start_select_t

List of Signal MUX for Trigger Based Measurement Start.

typedef enum _sfa_trigger_end_select sfa_trigger_end_select_t

List of Signal MUX for Trigger Based Measurement End.

typedef enum _sfa_trigger_start_polarity sfa_trigger_start_polarity_t

List of Trigger Start Polarity.

typedef enum _sfa_trigger_end_polarity sfa_trigger_end_polarity_t

List of Trigger End Polarity.

```
typedef struct _sfa_init_config sfa_config_t
```

Structure with setting to initialize the SFA module.

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

```
SFA__CUT__CLK__Enable(val)
```

```
struct _sfa_init_config
```

```
#include <fsl_sfa.h> Structure with setting to initialize the SFA module.
```

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

Public Members

```
sfa_measurement_mode_t mode
```

measurement mode

```
sfa_cut_select_t cutSelect
```

Select clock connected to the clock under test counter

```
sfa_ref_select_t refSelect
```

Select REF connected the bus clock

```
uint8_t prediv
```

Integer divide of the Input CUT signal

```
sfa_trigger_start_select_t trigStart
```

Select the signal will be used to end a trigger based measurement

```
sfa_trigger_start_polarity_t startPolarity
```

Select the polarity of the start trigger signal

```
sfa_trigger_end_select_t trigEnd
```

Select the signal will be used to commence a trigger based measurement

```
sfa_trigger_end_polarity_t endPolarity
```

Select the polarity of the end trigger signal

```
bool enableTrigMeasurement
```

false: The measurement will start by default with a dummy write to the CUT counter;
true : The measurement will start after receiving a dummy write to the REF_CNT followed by receiving the trigger edge

```
bool enableCUTPin
```

Control the connection of the clock under test to an external pin.

```
uint32_t refTarget
```

Reference counter target counts

```
uint32_t cutTarget
```

CUT counter target counts

2.66 SMSCM: Secure Miscellaneous System Control Module

FSL_MSCM_DRIVER_VERSION

SMSCM driver version 2.0.0.

enum _smscm_debug

SMSCM debug enable type.

Values:

enumerator kSMSCM_InvasiveDebug

enumerator kSMSCM_SecureInvasiveDebug

enumerator kSMSCM_NonInvasiveDebug

enumerator kSMSCM_SecureNonInvasiveDebug

enumerator kSMSCM_AltInvasiveDebug

enumerator kSMSCM_AltDebug

enum _smscm_mem

SMSCM On-Chip Memory Descriptor Register.

Values:

enumerator kSMSCM_Mem0

enumerator kSMSCM_Mem2

enumerator kSMSCM_Mem3

enumerator kSMSCM_Mem5

enum _smscm_ecc_ctrl

SMSCM ECC control type of On-Chip Memory.

Values:

enumerator kSMSCM_EccDisable

enumerator kSMSCM_EccEnableOnWrite

enumerator kSMSCM_EccEnableOnRead

enumerator kSMSCM_EccEnableOnWriteAndRead

typedef enum _smscm_mem smscm_mem_t

SMSCM On-Chip Memory Descriptor Register.

typedef enum _smscm_ecc_ctrl smscm_ecc_ctrl_t

SMSCM ECC control type of On-Chip Memory.

typedef struct smscm_ecc_fault_attr smscm_ecc_fault_attr_t

SMSCM attribute.

static inline void SMSCM_EnableDebug(SMSCM_Type *base, uint32_t debugToEnable)

Enable the Debug function. DebugToEnable could be bitwise OR of _smscm_debug.

Parameters

- base – SMSCM peripheral address.
- debugToEnable – debug enable type.

static inline void SMSCM_DisableDebug(SMSCM_Type *base, uint32_t debugToDisable)
Disables the Debug function. DebugToDisable could be bitwise OR of _smscm_debug.

Parameters

- base – SMSCM peripheral address.
- debugToDisable – debug disable type.

static inline void SMSCM_DebugLock(SMSCM_Type *base)
Lock the debug function.

Parameters

- base – SMSCM peripheral base address.

static inline uint32_t SMSCM_GetSecurityCount(SMSCM_Type *base)
Get value in Security Counter Register (SCTR).

Parameters

- base – SMSCM peripheral base address.

Returns

SMSCM SCTR value.

static inline void SMSCM_SetSecurityCount(SMSCM_Type *base, uint32_t val)
Set value in Security Counter Register (SCTR).

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to set.

static inline void SMSCM_IncreaseSecurityCount(SMSCM_Type *base, uint32_t val)
Write value to be plused in Security Counter Register (SCTR).

The entire contents of the write data word are added to the security counter, and next-state
 $SCTR = \text{current-state } SCTR + \text{DATA32}$.

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to plus.

static inline void SMSCM_DecreaseSecurityCount(SMSCM_Type *base, uint32_t val)
Write value to be minused in Security Counter Register (SCTR).

The entire contents of the write data word are added to the security counter, and next-state
 $SCTR = \text{current-state } SCTR - \text{DATA32}$.

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to be minused.

static inline void SMSCM_IncreaseSecurityCountBy1(SMSCM_Type *base)
Increase security counter register by 1.

Parameters

- base – SMSCM peripheral base address.

static inline void SMSCM_DecreaseSecurityCountBy1(SMSCM_Type *base)
Decrease security counter register by 1.

Parameters

- base – SMSCM peripheral base address.

static inline void SMSCM_LockMemControlReg(SMSCM_Type *base, *smscm_mem_t* mem)

Lock the on-chip memory descriptor. This register bit provides a mechanism to “lock” the configuration state defined by OCMDRn[11:0]. Once asserted, attempted writes to the OCM-DRn[11:0] register are ignored until the next reset clears the flag.

Parameters

- base – SMSCM peripheral address.
- mem – Select OCMDRn to enable read-only mode.

static inline void SMSCM_EnableFlashCache(SMSCM_Type *base, bool enable)

Enable or disable the on-chip memory flash cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashInstructionCache(SMSCM_Type *base, bool enable)

Enable flash instruction cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashDataCache(SMSCM_Type *base, bool enable)

Enable flash data cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_ClearFlashCache(SMSCM_Type *base)

Clear the on-chip memory flash cache.

Parameters

- base – SMSCM peripheral address.

static inline void SMSCM_LockFlashIFR1(SMSCM_Type *base)

Lock IFR1 by flash controller.

Parameters

- base – SMSCM peripheral address.

static inline void SMSCM_EnableFlashSpeculate(SMSCM_Type *base, bool enable)

SMSCM Flash Speculate enable.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableDataPrefetch(SMSCM_Type *base, bool enable)

SMSCM Data Prefetch enable.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

```
static inline void SMSCM_EnableFlashDataNonCorrectableBusError(SMSCM_Type *base, bool enable)
```

Disable non-correctable bus errors on flash data fetches.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

```
static inline void SMSCM_EnableFlashInstructionNonCorrectableBusError(SMSCM_Type *base, bool enable)
```

Disable non-correctable bus errors on flash instruction fetches.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

```
static inline void SMSCM_SetMemEccControl(SMSCM_Type *base, smscm_mem_t mem, smscm_ecc_ctrl_t eccCtrl)
```

Select ecc control type in OCMDRn.

Parameters

- base – SMSCM peripheral address.
- mem – Select OCMDRn.
- eccCtrl – Select ecc control type.

```
static inline void SMSCM_EnableEccReport(SMSCM_Type *base)
```

Enable RAM ECC 1 bit and non-correctable reporting.

Parameters

- base – SMSCM peripheral base address.

```
static inline bool SMSCM_GetEccValid(SMSCM_Type *base)
```

Get the ECC location valid states.

Parameters

- base – SMSCM peripheral base address.

Returns

State of ECC Error Location field.

```
static inline uint8_t SMSCM_GetEccLocation(SMSCM_Type *base)
```

Get the ECC location.

Parameters

- base – SMSCM peripheral base address.

Returns

ECC fault location.

```
void SMSCM_ClearEccError(SMSCM_Type *base, uint8_t errLocation)
```

Clear each 1-bit correctable or non-correctable error.

Parameters

- base – SMSCM peripheral address.
- errLocation – ECC Error Location

```
static inline uint32_t SMSCM_GetEccAddress(SMSCM_Type *base)
```

Get the ECC fault address.

Parameters

- base – SMSCM peripheral base address.

Returns

ECC fault address.

```
void SMSCM_GetEccAttribute(SMSCM_Type *base, smscm_ecc_fault_attr_t *eccAttribute)
```

Get ECC attribute.

Parameters

- base – SMSCM peripheral address.
- eccAttribute – Ecc attribute.

```
static inline uint32_t SMSCM_GetEccFaultDataHigh(SMSCM_Type *base)
```

Get ECC Fault Data High.

This read-only field specifies the upper 32-bit read data word (data[63:32]) from the last captured ECCevent. For ECC events that occur in 32-bit RAMs, this 32-bit field will return 32'h0.

Parameters

- base – SMSCM peripheral base address.

Returns

The higher 32-bit read data word.

```
static inline uint32_t SMSCM_GetEccFaultDataLow(SMSCM_Type *base)
```

Get ECC Fault Data Low.

Parameters

- base – SMSCM peripheral base address.

Returns

The lower 32-bit read data word.

```
struct smscm_ecc_fault_attr
```

#include <fsl_smscm.h> SMSCM attribute.

2.67 SPC: System Power Control driver

SPC status enumeration.

Values:

enumerator kStatus_SPC_Busy

The SPC instance is busy executing any type of power mode transition.

enumerator kStatus_SPC_DCDCLowDriveStrengthIgnore

DCDC Low drive strength setting be ignored for LVD/HVD enabled.

enumerator kStatus_SPC_DCDCPulseRefreshModeIgnore

DCDC Pulse Refresh Mode drive strength setting be ignored for LVD/HVD enabled.

enumerator kStatus_SPC_SYSLDOOverDriveVoltageFail

SYS LDO regulate to Over drive voltage failed for SYS LDO HVD must be disabled.

enumerator kStatus_SPC_SYSLDOLowDriveStrengthIgnore

SYS LDO Low driver strength setting be ignored for LDO LVD/HVD enabled.

enumerator kStatus_SPC_CORELDOLowDriveStrengthIgnore

CORE LDO Low driver strength setting be ignored for LDO LVD/HVD enabled.

enumerator kStatus_SPC_CORELDOLowVoltageWrong

Core LDO voltage is wrong.

enumerator kStatus_SPC_CORELDOLowVoltageSetFail

Core LDO voltage set fail.

enumerator kStatus_SPC_BandgapModeWrong

Selected Bandgap Mode wrong.

enum _spc_voltage_detect_flags

Voltage Detect Status Flags.

Values:

enumerator kSPC_IOVDDHighVoltageDetectFlag

IO VDD High-Voltage detect flag.

enumerator kSPC_SystemVDDHighVoltageDetectFlag

System VDD High-Voltage detect flag.

enumerator kSPC_CoreVDDHighVoltageDetectFlag

Core VDD High-Voltage detect flag.

enumerator kSPC_IOVDDLowVoltageDetectFlag

IO VDD Low-Voltage detect flag.

enumerator kSPC_SystemVDDLowVoltageDetectFlag

System VDD Low-Voltage detect flag.

enumerator kSPC_CoreVDDLowVoltageDetectFlag

Core VDD Low-Voltage detect flag.

enum _spc_power_domains

SPC power domain isolation status.

Values:

enumerator kSPC_MAINPowerDomainRetain

Peripherals and IO pads retain in MAIN Power Domain.

enumerator kSPC_WAKEPowerDomainRetain

Peripherals and IO pads retain in WAKE Power Domain.

enumerator kSPC_2P4GPowerDomainRetain

Peripherals and IO pads retain in 2.4G Power Domain.

enum _spc_power_domain_id

The enumeration of spc power domain, the connected power domain is chip specific, please refer to chip's RM for details.

Values:

enumerator kSPC_PowerDomain0

Power domain0, the connected power domain is chip specific.

enumerator kSPC_PowerDomain1

Power domain1, the connected power domain is chip specific.

enumerator kSPC_PowerDomain2

Power domain2, the connected power domain is chip specific.

enum _spc_power_domain_low_power_mode

The enumeration of Power domain's low power mode.

Values:

enumerator kSPC_SleepWithSYSClockRunning

Power domain request SLEEP mode with SYS clock running.

enumerator kSPC_SleepWithSysClockOff

Power domain request SLEEP mode with SYS clock off.

enumerator kSPC_DeepSleepSysClockOff

Power domain request DEEP SLEEP mode with SYS clock off.

enumerator kSPC_PowerDownWithSysClockOff

Power domain request POWER DOWN mode with SYS clock off.

enumerator kSPC_DeepPowerDownWithSysClockOff

Power domain request DEEP POWER DOWN mode with SYS clock off.

enum _spc_lowPower_request_pin_polarity

SPC low power request output pin polarity.

Values:

enumerator kSPC_HighTruePolarity

Control the High Polarity of the Low Power Request Pin.

enumerator kSPC_LowTruePolarity

Control the Low Polarity of the Low Power Request Pin.

enum _spc_lowPower_request_output_override

SPC low power request output override.

Values:

enumerator kSPC_LowPowerRequestNotForced

Not Forced.

enumerator kSPC_LowPowerRequestReserved

Reserved.

enumerator kSPC_LowPowerRequestForcedLow

Forced Low (Ignore LowPower request output polarity setting.)

enumerator kSPC_LowPowerRequestForcedHigh

Forced High (Ignore LowPower request output polarity setting.)

enum _spc_bandgap_mode

SPC Bandgap mode enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_BandgapDisabled

Bandgap disabled.

enumerator kSPC_BandgapEnabledBufferDisabled

Bandgap enabled with Buffer disabled.

enumerator kSPC_BandgapEnabledBufferEnabled

Bandgap enabled with Buffer enabled.

enumerator kSPC_BandgapReserved
Reserved.

enum _spc_dcdc_voltage_level

DCDC regulator voltage level enumeration in Active mode or Low Power Mode.

Values:

enumerator kSPC_DCDC_SafeModeVoltage
DCDC VDD Regulator regulate to Safe-Mode Voltage.

enumerator kSPC_DCDC_NormalVoltage
DCDC VDD Regulator regulate to Normal Voltage.

enumerator kSPC_DCDC_MidVoltage
DCDC VDD Regulator regulate to Mid Voltage.

enumerator kSPC_DCDC_LowUnderVoltage
DCDC VDD Regulator regulate to Low Under Voltage.

enum _spc_dcdc_drive_strength

DCDC regulator Drive Strength enumeration in Active mode or Low Power Mode.

Values:

enumerator kSPC_DCDC_PulseRefreshMode
DCDC VDD Regulator Drive Strength set to Pulse Refresh Mode. This enum member is only useful for Low Power Mode config.

enumerator kSPC_DCDC_LowDriveStrength
DCDC VDD regulator Drive Strength set to low.

enumerator kSPC_DCDC_NormalDriveStrength
DCDC VDD regulator Drive Strength set to Normal.

enumerator kSPC_DCDC_Reserved
Reserved.

enum _spc_sys_ldo_voltage_level

SYS LDO regulator voltage level enumeration in Active mode.

Values:

enumerator kSPC_SysLDO_NormalVoltage
SYS LDO VDD Regulator regulate to Normal Voltage(1.8V).

enumerator kSPC_SysLDO_OverDriveVoltage
SYS LDO VDD Regulator regulate to Over Drive Voltage(2.5V).

enum _spc_sys_ldo_drive_strength

SYS LDO regulator Drive Strength enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_SysLDO_LowDriveStrength
SYS LDO VDD regulator Drive Strength set to low.

enumerator kSPC_SysLDO_NormalDriveStrength
SYS LDO VDD regulator Drive Strength set to Normal.

enum _spc_core_ldo_voltage_level

Core LDO regulator voltage level enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_CoreLDO_NormalVoltage
Core LDO VDD regulator regulate to Normal Voltage.

enumerator kSPC_CoreLDO_MidDriveVoltage
Core LDO VDD regulator regulate to Mid Drive Voltage.

enumerator kSPC_CoreLDO_UnderDriveVoltage
Core LDO VDD regulator regulate to Under Drive Voltage.

enumerator kSPC_CoreLDO_SafeModeVoltage
Core LDO VDD regulator regulate to Safe-Mode Voltage.

enum _spc_core_ldo_drive_strength
CORE LDO VDD regulator Drive Strength enumeration in Low Power mode.

Values:

enumerator kSPC_CoreLDO_LowDriveStrength
Core LDO VDD regulator Drive Strength set to low.

enumerator kSPC_CoreLDO_NormalDriveStrength
Core LDO VDD regulator Drive Strength set to Normal.

enum _spc_low_voltage_level_select
System/IO VDD Low-Voltage Level Select.

Values:

enumerator kSPC_LowVoltageNormalLevel
Trip point set to Normal level.

enumerator kSPC_LowVoltageSafeLevel
Trip point set to Safe level.

enum _spc_sram_operat_voltage
SRAM operate voltage enumeration.

Values:

enumerator kSPC_SRAM_OperatVoltage1P0V
SRAM operate voltage set to 1.0V.

enumerator kSPC_SRAM_OperatVoltage1P1V
SRAM operate voltage set to 1.1V.

enum _spc_hp_request_override_option
The enumeration of high power request override option.

Values:

enumerator kSPC_HpRequestOverrideDisable
Disable high power request override feature.

enumerator kSPC_HpRequestOverride0
Enable high power request override feature and force value as 0.

enumerator kSPC_HpRequestOverride1
Enable high power request override feature and force value as 1.

enum _spc_vdd_core_glitch_ripple_counter_select
Used to select output of 4-bit ripple counter is used to monitor a glitch on VDD core.

Values:

enumerator `kSPC_selectBit0Of4bitRippleCounter`

Select bit-0 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator `kSPC_selectBit1Of4bitRippleCounter`

Select bit-1 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator `kSPC_selectBit2Of4bitRippleCounter`

Select bit-2 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator `kSPC_selectBit3Of4bitRippleCounter`

Select bit-3 of 4-bit Ripple Counter to detect glitch on VDD Core.

typedef enum `_spc_power_domain_id` `spc_power_domain_id_t`

The enumeration of spc power domain, the connected power domain is chip specific, please refer to chip's RM for details.

typedef enum `_spc_power_domain_low_power_mode` `spc_power_domain_low_power_mode_t`

The enumeration of Power domain's low power mode.

typedef enum `_spc_lowPower_request_pin_polarity` `spc_lowpower_request_pin_polarity_t`

SPC low power request output pin polarity.

typedef enum `_spc_lowPower_request_output_override` `spc_lowpower_request_output_override_t`

SPC low power request output override.

typedef enum `_spc_bandgap_mode` `spc_bandgap_mode_t`

SPC Bandgap mode enumeration in Active mode or Low Power mode.

typedef enum `_spc_dcdc_voltage_level` `spc_dcdc_voltage_level_t`

DCDC regulator voltage level enumeration in Active mode or Low Power Mode.

typedef enum `_spc_dcdc_drive_strength` `spc_dcdc_drive_strength_t`

DCDC regulator Drive Strength enumeration in Active mode or Low Power Mode.

typedef enum `_spc_sys_ldo_voltage_level` `spc_sys_ldo_voltage_level_t`

SYS LDO regulator voltage level enumeration in Active mode.

typedef enum `_spc_sys_ldo_drive_strength` `spc_sys_ldo_drive_strength_t`

SYS LDO regulator Drive Strength enumeration in Active mode or Low Power mode.

typedef enum `_spc_core_ldo_voltage_level` `spc_core_ldo_voltage_level_t`

Core LDO regulator voltage level enumeration in Active mode or Low Power mode.

typedef enum `_spc_core_ldo_drive_strength` `spc_core_ldo_drive_strength_t`

CORE LDO VDD regulator Drive Strength enumeration in Low Power mode.

typedef enum `_spc_low_voltage_level_select` `spc_low_voltage_level_select_t`

System/IO VDD Low-Voltage Level Select.

typedef enum `_spc_sram_operat_voltage` `spc_sram_operat_voltage_t`

SRAM operate voltage enumeration.

typedef struct `_spc_lowpower_request_config` `spc_lowpower_request_config_t`

Low Power Request output pin configuration.

typedef struct `_spc_intergrated_power_switch_config` `spc_intergrated_power_switch_config_t`

Integrated power switch configuration.

Note: Legacy structure, will be removed.

```
typedef struct _spc_active_mode_core_ldo_option spc_active_mode_core_ldo_option_t
    Core LDO regulator options in Active mode.
typedef struct _spc_active_mode_sys_ldo_option spc_active_mode_sys_ldo_option_t
    System LDO regulator options in Active mode.
typedef struct _spc_active_mode_dcdc_option spc_active_mode_dcdc_option_t
    DCDC regulator options in Active mode.
typedef struct _spc_lowpower_mode_core_ldo_option spc_lowpower_mode_core_ldo_option_t
    Core LDO regulator options in Low Power mode.
typedef struct _spc_lowpower_mode_sys_ldo_option spc_lowpower_mode_sys_ldo_option_t
    System LDO regulator options in Low Power mode.
typedef struct _spc_lowpower_mode_dcdc_option spc_lowpower_mode_dcdc_option_t
    DCDC regulator options in Low Power mode.
typedef struct _spc_voltage_detect_option spc_voltage_detect_option_t
    CORE/SYS/IO VDD Voltage Detect options.
typedef struct _spc_dcdc_burst_config spc_dcdc_burst_config_t
    DCDC Burst configuration.
typedef struct _spc_core_voltage_detect_config spc_core_voltage_detect_config_t
    Core Voltage Detect configuration.
typedef struct _spc_system_voltage_detect_config spc_system_voltage_detect_config_t
    System Voltage Detect Configuration.
typedef struct _spc_io_voltage_detect_config spc_io_voltage_detect_config_t
    IO Voltage Detect Configuration.
typedef struct _spc_active_mode_regulators_config spc_active_mode_regulators_config_t
    Active mode configuration.
typedef struct _spc_lowpower_mode_regulators_config spc_lowpower_mode_regulators_config_t
    Low Power Mode configuration.
typedef enum _spc_hp_request_override_option spc_hp_override_request_option_t
    The enumeration of high power request override option.
typedef spc_active_mode_dcdc_option_t spc_hp_mode_dcdc_option_t
typedef spc_active_mode_sys_ldo_option_t spc_hp_mode_sys_ldo_option_t
typedef spc_active_mode_core_ldo_option_t spc_hp_mode_core_ldo_option_t
typedef spc_active_mode_regulators_config_t spc_hp_mode_regulators_config_t
typedef enum _spc_vdd_core_glitch_ripple_counter_select
    spc_vdd_core_glitch_ripple_counter_select_t
        Used to select output of 4-bit ripple counter is used to monitor a glitch on VDD core.
typedef struct _spc_vdd_core_glitch_detector_config spc_vdd_core_glitch_detector_config_t
    The configuration of VDD Core glitch detector.
```

SPC_BUSY_TIMEOUT

Max loops to wait for SPC to stop being busy.

The BUSY bitfield will be set when the SPC performs any kind of power mode transition in active mode or any chip low power mode. You need to wait until this flag is cleared before changing the power mode configuration registers. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until completion.

SPC_SRAM_ACK_TIMEOUT

Max loops to wait for SPC to stop being busy.

When changing SRAM voltage, need to wait for SRAM voltage update request acknowledgment. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until completion.

SPC_DCDC_ACK_TIMEOUT

Max loops to wait for DCDC burst completed.

When the DCDC burst is requested, it is necessary to wait for the DCDC burst to complete. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until it completes.

SPC_HP_CNFG_CTRL_OVERRIDE_OPT_MASK**SPC_HP_CNFG_CTRL_OVERRIDE_OPT_SHIFT****SPC_HP_CNFG_CTRL_OVERRIDE_OPT(x)****uint8_t SPC_GetPeriphIOIsolationStatus(SPC_Type *base)**

Gets Isolation status for each power domains.

This function gets the status which indicates whether certain peripheral and the IO pads are in a latched state as a result of having been in POWERDOWN mode.

Parameters

- base – SPC peripheral base address.

Returns

Current isolation status for each power domains. See `_spc_power_domains` for details.

static inline void SPC_ClearPeriphIOIsolationFlag(SPC_Type *base)

Clears peripherals and I/O pads isolation flags for each power domains.

This function clears peripherals and I/O pads isolation flags for each power domains. After recovering from the POWERDOWN mode, user must invoke this function to release the I/O pads and certain peripherals to their normal run mode state. Before invoking this function, user must restore chip configuration in particular pin configuration for enabled WUU wakeup pins.

Parameters

- base – SPC peripheral base address.

static inline bool SPC_GetBusyStatusFlag(SPC_Type *base)

Gets SPC busy status flag.

This function gets SPC busy status flag. When SPC executing any type of power mode transition in ACTIVE mode or any of the SOC low power mode, the SPC busy status flag is set and this function returns true. When changing CORE LDO voltage level and DCDC voltage level in ACTIVE mode, the SPC busy status flag is set and this function return true.

Parameters

- base – SPC peripheral base address.

Returns

Ack busy flag. true - SPC is busy. false - SPC is not busy.

static inline bool SPC_CheckLowPowerRequest(SPC_Type *base)

Checks system low power request.

Note: Only when all power domains request low power mode entry, the result of this function is true. That means when all power domains request low power mode entry, the SPC regulators will be controlled by LP_CFG register.

Parameters

- `base` – SPC peripheral base address.

Returns

The system low power request check result.

- **true** All power domains have requested low power mode and SPC has entered a low power state and power mode configuration are based on the LP_CFG configuration register.
- **false** SPC in active mode and ACTIVE_CFG register control system power supply.

```
static inline void SPC_ClearLowPowerRequest(SPC_Type *base)
```

Clears system low power request, set SPC in active mode.

Parameters

- `base` – SPC peripheral base address.

```
static inline bool SPC_CheckPowerSwitchState(SPC_Type *base)
```

Checks power switch state.

Parameters

- `base` – SPC peripheral base address.

Returns

The state(ON/OFF) of power switch.

- **true** Indicates the power switch is ON.
- **false** Indicates the power switch is OFF.

```
spc_power_domain_low_power_mode_t SPC_GetPowerDomainLowPowerMode(SPC_Type *base,  
                                                                    spc_power_domain_id_t  
                                                                    powerDomainId)
```

Gets selected power domain's requested low power mode.

Parameters

- `base` – SPC peripheral base address.
- `powerDomainId` – Power Domain Id, please refer to `spc_power_domain_id_t`.

Returns

The selected power domain's requested low power mode, please refer to `spc_power_domain_low_power_mode_t`.

```
static inline bool SPC_CheckPowerDomainLowPowerRequest(SPC_Type *base,  
                                                       spc_power_domain_id_t  
                                                       powerDomainId)
```

Checks power domain's low power request.

Parameters

- `base` – SPC peripheral base address.
- `powerDomainId` – Power Domain Id, please refer to `spc_power_domain_id_t`.

Returns

The result of power domain's low power request.

- **true** The selected power domain requests low power mode entry.
- **false** The selected power domain does not request low power mode entry.

```
static inline void SPC_ClearPowerDomainLowPowerRequestFlag(SPC_Type *base,  
                                                         spc_power_domain_id_t  
                                                         powerDomainId)
```

Clears selected power domain's low power request flag.

Parameters

- base – SPC peripheral base address.
- powerDomainId – Power Domain Id, please refer to spc_power_domain_id_t.

```
void SPC_SetLowPowerRequestConfig(SPC_Type *base, const spc_lowpower_request_config_t  
                                *config)
```

Configs Low power request output pin.

This function config the low power request output pin

Parameters

- base – SPC peripheral base address.
- config – Pointer the spc_lowpower_request_config_t structure.

```
static inline void SPC_SoftwareGatePowerSwitch(SPC_Type *base, bool gate)
```

Gates/Un-gates power switch in software mode.

Parameters

- base – SPC peripheral base address.
- gate – Used to gate/ungate power switch
 - **true** The power switch will be gated.
 - **false** The power switch will be un-gated.

```
static inline void SPC_PowerModeControlPowerSwitch(SPC_Type *base)
```

Gates power switch at low power modes entry, and un-gates power switch at low power mode wakeup.

Parameters

- base – SPC peripheral base address.

```
static inline void SPC_SetWakeUpValue(SPC_Type *base, uint32_t data)
```

Set the address of the function/image to be executed if chip wake from power down or deep power down mode.

Note: Data written by this function is used by BootROM to quickly restore ARM Core context, or to switch execution to a defined address in Flash/SRAM on WakeUp.

Note: The first word must be SP, and the second word must be PC.

Note: Please remember to calculate the CRC value of the first 48 bytes of image/function and save the result to REGFILE1->REG[0]. The BootROM will check this CRC value, if authenticated successfully then the image/function will be executed.

Parameters

- `base` – SPC peripheral base address.
- `data` – The address of the function/image to be executed if wakeup from low power mode.

static inline uint32_t SPC_GetWakeUpValue(SPC_Type *base)

Gets back the WakeUp value.

Parameters

- `base` – SPC peripheral base address.

Returns

The WakeUp value.

static inline bool SPC_CheckHPCfgSelected(SPC_Type *base)

Check if HP_CFG selected as active configuration register.

Parameters

- `base` – SPC peripheral base address.

Return values

- `false` – ACTIVE_CFG selected as the active configuration register.
- `true` – HP_CFG selected as the active configuration register.

static inline void SPC_EnableHighPowerRequest(SPC_Type *base, bool enable)

Enable/disable high power request feature.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Used to specify enable/disable the high power request feature:
 - **true** Enable high power request;
 - **false** Disable high power request.

static inline void SPC_OverrideHighPowerRequest(SPC_Type *base,
sps_hp_override_request_option_t opt)

Override high power request manually.

Parameters

- `base` – SPC peripheral base address.
- `opt` – Specify the option of high power override request, please refer to `sps_hp_override_request_option_t`.

static inline sps_core_ldo_voltage_level_t SPC_GetHighPowerModeCoreLDOVDDVoltageLevel(SPC_Type *base)

Get voltage level of CORE LDO in high power mode.

Parameters

- `base` – SPC peripheral base address.

Returns

The voltage level of CORE LDO in high power mode, please refer to `spc_core_ldo_voltage_level_t`.

```
static inline spc_bandgap_mode_t SPC_GetHighPowerModeBandgapMode(SPC_Type *base)
```

Get bandgap mode in high power mode.

Parameters

- `base` – SPC peripheral base address.

Returns

The bandgap mode in high power mode, please refer to `spc_bandgap_mode_t`.

```
static inline uint32_t SPC_GetHighPowerModeVoltageDetectStatus(SPC_Type *base)
```

Get enabled state of all voltage detectors.

Parameters

- `base` – SPC peripheral base address.

Returns

All enabled status of all voltage detectors, 1b1 means the corresponding voltage detector is enabled.

```
status_t SPC_SetHighPowerModeBandgapModeConfig(SPC_Type *base, spc_bandgap_mode_t  
mode)
```

Set bandgap mode in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `mode` – Specify the bandgap mode in high power mode.

Return values

- `kStatus_SPC_BandgapModeWrong` – The Bandgap can not be disabled in high power mode.
- `kStatus_Success` – Config Bandgap mode in high power mode successful.

```
static inline void SPC_EnableHighPowerModeCMPBandgapBuffer(SPC_Type *base, bool enable)
```

Enable/disable CMP buffer in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Used to enable/disable CMP buffer:
 - **true** Enable CMP buffer in high power mode;
 - **false** Disable CMP buffer in high power mode.

```
static inline void SPC_DisableHighPowerModeVddCoreGlitchDetect(SPC_Type *base, bool disable)
```

Disable/enable VDD Core Glitch detect feature in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `disable` – Used to disable/enable VDD Core Glitch detect feature:
 - **true** Disable VDD Core Glitch detect feature;
 - **false** Enable VDD Core Glitch detect feature.

```
status_t SPC_SetHighPowerModeCoreLDORegulatorConfig(SPC_Type *base, const
                                                    spc_hp_mode_core_ldo_option_t
                                                    *option)
```

Configure CORE LDO regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the CORE LDO regulator configuration, please refer to `spc_hp_mode_core_ldo_option_t`.

Return values

- `kStatus_Success` – Config Core LDO regulator in High power mode successful.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_CORELDOLowDriveStrengthIgnore` – If any voltage detect enabled, core_ldo's drive strength can not set to low.
- `kStatus_SPC_CORELDOPVoltageWrong` – The selected voltage level in high power mode is not allowed.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

```
status_t SPC_SetHighPowerModeSystemLDORegulatorConfig(SPC_Type *base, const
                                                    spc_hp_mode_sys_ldo_option_t
                                                    *option)
```

Configure System LDO regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the SYSTEM LDO regulator configuration, please refer to `spc_hp_mode_sys_ldo_option_t`.

Return values

- `kStatus_Success` – Config System LDO regulator in Active power mode successful.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_SYSLDOOverDriveVoltageFail` – Fail to regulator to Over Drive Voltage.
- `kStatus_SPC_SYSLDOLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

```
status_t SPC_SetHighPowerModeDCDCRegulatorConfig(SPC_Type *base, const
                                                    spc_hp_mode_dcdc_option_t *option)
```

Configure DCDC regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the DCDC regulator configuration, please refer to `spc_hp_mode_dcdc_option_t`.

Return values

- `kStatus_Success` – Config DCDC regulator in Active power mode successful.

- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_DCDCLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`status_t SPC_SetHighPowerModeRegulatorsConfig(SPC_Type *base, const
spc_hp_mode_regulators_config_t *config)`

Set configuration of regulators in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `config` – Pointer to the regulator configuration, please refer to `spc_hp_mode_regulators_config_t`.

Return values

- `kStatus_Success` – Config regulators in High power mode successful.
- `kStatus_SPC_BandgapModeWrong` – The bandgap mode setting in high power mode is wrong.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_CORELDOVoltageWrong` – The selected voltage level in high mode is not allowed.
- `kStatus_SPC_SYSLDOOverDriveVoltageFail` – Fail to regulator to Over Drive Voltage.
- `kStatus_SPC_SYSLDOLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_SPC_DCDCLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.

`static inline void SPC_EnableHighPowerModeCoreLowVoltageDetect(SPC_Type *base, bool
enable)`

Enable/disable low voltage detect for VDD_CORE in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Used to enable/disable low voltage detect feature for VDD_CORE in high power mode:
 - **true** Enable low voltage detect feature for VDD_CORE in high power mode;
 - **false** Disable low voltage detect feature for VDD_CORE in high power mode.

`static inline void SPC_EnableHighPowerModeCoreHighVoltageDetect(SPC_Type *base, bool
enable)`

Enable/disable high voltage detect for VDD_CORE in high power mode.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Used to enable/disable high voltage detect feature for VDD_CORE in high power mode:

- **true** Enable low voltage detect feature for VDD_CORE in high power mode;
- **false** Disable low voltage detect feature for VDD_CORE in high power mode.

```
static inline void SPC__EnableHighPowerModeSystemLowVoltageDetect(SPC_Type *base, bool enable)
```

Enable/disable low voltage detect for VDD_SYS in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable low voltage detect feature for VDD_SYS in high power mode:
 - **true** Enable low voltage detect feature for VDD_SYS in high power mode;
 - **false** Disable low voltage detect feature for VDD_SYS in high power mode.

```
static inline void SPC__EnableHighPowerModeSystemHighVoltageDetect(SPC_Type *base, bool enable)
```

Enable/disable high voltage detect for VDD_SYS in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable high voltage detect feature for VDD_SYS in high power mode:
 - **true** Enable high voltage detect feature for VDD_SYS in high power mode;
 - **false** Disable high voltage detect feature for VDD_SYS in high power mode.

```
static inline void SPC__EnableHighPowerModeIOLowVoltageDetect(SPC_Type *base, bool enable)
```

Enable/disable low voltage detect for VDD_IO_ABC in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable low voltage detect feature for VDD_IO_ABC in high power mode:
 - **true** Enable low voltage detect feature for VDD_IO_ABC in high power mode;
 - **false** Disable low voltage detect feature for VDD_IO_ABC in high power mode.

```
static inline void SPC__EnableHighPowerModeIOHighVoltageDetect(SPC_Type *base, bool enable)
```

Enable/disable high voltage detect for VDD_IO_ABC in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable high voltage detect feature for VDD_IO_ABC in high power mode:
 - **true** Enable high voltage detect feature for VDD_IO_ABC in high power mode;
 - **false** Disable high voltage detect feature for VDD_IO_ABC in high power mode.

```
static inline spc_core_ldo_voltage_level_t SPC_GetActiveModeCoreLDOVDDVoltageLevel(SPC_Type  
                                                                                   *base)
```

Gets CORE LDO VDD Regulator Voltage level.

This function returns the voltage level of CORE LDO Regulator in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

Voltage level of CORE LDO in type of *spc_core_ldo_voltage_level_t* enumeration.

```
static inline spc_bandgap_mode_t SPC_GetActiveModeBandgapMode(SPC_Type *base)
```

Gets the Bandgap mode in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

Bandgap mode in the type of *spc_bandgap_mode_t* enumeration.

```
static inline uint32_t SPC_GetActiveModeVoltageDetectStatus(SPC_Type *base)
```

Gets all voltage detectors status in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

All voltage detectors status in Active mode.

```
void SPC_SetActiveModeIntegratedPowerSwitchConfig(SPC_Type *base, const  
                                                    spc_intergrated_power_switch_config_t  
                                                    *config)
```

Configs Integrated power switch in active mode.

Note: Legacy API and will be removed.

Parameters

- base – SPC peripheral base address.
- config – Pointer to *spc_intergrated_power_switch_config_t* pointer.

```
status_t SPC_SetActiveModeBandgapModeConfig(SPC_Type *base, spc_bandgap_mode_t mode)
```

Configs Bandgap mode in Active mode.

Note: In active mode, because CORELDO_VDD_DS is reserved and set to Normal, so it is impossible to disable Bandgap in active mode

Parameters

- base – SPC peripheral base address.
- mode – The Bandgap mode be selected.

Return values

- *kStatus_SPC_BandgapModeWrong* – The Bandgap can not be disabled in active mode.

- `kStatus_Success` – Config Bandgap mode in Active power mode successful.

`static inline void SPC_EnableActiveModeCMPBandgapBuffer(SPC_Type *base, bool enable)`

Enables/Disable the CMP Bandgap Buffer in Active mode.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Enable/Disable CMP Bandgap buffer. `true` - Enable Buffer Stored Reference voltage to CMP. `false` - Disable Buffer Stored Reference voltage to CMP.

`static inline void SPC_SetActiveModeVoltageTrimDelay(SPC_Type *base, uint16_t delay)`

Sets the delay when the regulators change voltage level in Active mode.

Parameters

- `base` – SPC peripheral base address.
- `delay` – The number of SPC timer clock cycles.

`status_t SPC_SetActiveModeRegulatorsConfig(SPC_Type *base, const
 spc_active_mode_regulators_config_t *config)`

Configs regulators in Active mode.

This function provides the method to config all on-chip regulators in active mode.

Parameters

- `base` – SPC peripheral base address.
- `config` – Pointer to `spc_active_mode_regulators_config_t` structure.

Return values

- `kStatus_Success` – Config regulators in Active power mode successful.
- `kStatus_SPC_BandgapModeWrong` – The bandgap mode setting in Active mode is wrong.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_CORELDOVoltageWrong` – The selected voltage level in active mode is not allowed.
- `kStatus_SPC_SYSLDOOverDriveVoltageFail` – Fail to regulator to Over Drive Voltage.
- `kStatus_SPC_SYSLDOLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_SPC_DCDCLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`static inline void SPC_DisableActiveModeVddCoreGlitchDetect(SPC_Type *base, bool disable)`

Disable/Enable VDD Core Glitch Detect in Active mode.

Note: State of glitch detect disable feature will be ignored if bandgap is disabled and glitch detect hardware will be forced to OFF state.

Parameters

- `base` – SPC peripheral base address.

- **disable** – Used to disable/enable VDD Core Glitch detect feature.
 - **true** Disable VDD Core Low Voltage detect;
 - **false** Enable VDD Core Low Voltage detect.

```
void SPC_SetLowPowerModeIntegratedPowerSwitchConfig(SPC_Type *base, const
                                                    spc_intergrated_power_switch_config_t
                                                    *config)
```

Configs Integrated power switch in Low Power mode.

Note: Legacy API, will be removed.

Parameters

- **base** – SPC peripheral base address.
- **config** – Pointer to `spc_intergrated_power_switch_config_t` pointer.

```
static inline void SPC_EnableLowPowerModeVDDCWellBias(SPC_Type *base, bool enable)
```

Enables/Disables VDDC Well Bias in low power mode.

Parameters

- **base** – SPC peripheral base address.
- **enable** – Enable/Disable the VDDC Well Bias. **true** - Enable Vddc Well Bias.
false - Disable Vddc Well Bias.

```
static inline spc_core_ldo_drive_strength_t SPC_GetLowPowerCoreLDOVDDDriveStrength(SPC_Type
                                                                                      *base)
```

Gets CORE LDO VDD Drive Strength for Low Power modes.

Parameters

- **base** – SPC peripheral base address.

Returns

The CORE LDO's VDD Drive Strength.

```
static inline spc_core_ldo_voltage_level_t SPC_GetLowPowerCoreLDOVDDVoltageLevel(SPC_Type
                                                                                      *base)
```

Gets the CORE LDO VDD Regulator Voltage Level for Low Power modes.

Parameters

- **base** – SPC peripheral base address.

Returns

The CORE LDO VDD Regulator's voltage level.

```
static inline spc_bandgap_mode_t SPC_GetLowPowerModeBandgapMode(SPC_Type *base)
```

Gets the Bandgap mode in Low Power mode.

Parameters

- **base** – SPC peripheral base address.

Returns

Bandgap mode in the type of `spc_bandgap_mode_t` enumeration.

```
static inline uint32_t SPC_GetLowPowerModeVoltageDetectStatus(SPC_Type *base)
```

Gets the status of all voltage detectors in Low Power mode.

Parameters

- **base** – SPC peripheral base address.

Returns

The status of all voltage detectors in low power mode.

```
static inline void SPC_EnableLowPowerModeLowPowerIREF(SPC_Type *base, bool enable)
```

Enables/Disables Low Power IREF in low power modes.

This function enables/disables Low Power IREF. Low Power IREF can only get disabled in Deep power down mode. In other low power modes, the Low Power IREF is always enabled.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Low Power IREF. true - Enable Low Power IREF for Low Power modes. false - Disable Low Power IREF for Deep Power Down mode.

```
status_t SPC_SetLowPowerModeBandgapmodeConfig(SPC_Type *base, spc_bandgap_mode_t mode)
```

Configs Bandgap mode in Low Power mode.

This function configs Bandgap mode in Low Power mode. IF user want to disable Bandgap while keeping any of the Regulator in Normal Driver Strength or if any of the High voltage detectors/Low voltage detectors are kept enabled, the Bandgap mode will be set as Bandgap Enabled with Buffer Disabled.

Note: This API shall be invoked following set HVDs/LVDs and regulators' driver strength.

Parameters

- base – SPC peripheral base address.
- mode – The Bandgap mode be selected.

Return values

- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in Low Power mode is wrong.
- kStatus_Success – Config Bandgap mode in Low Power power mode successful.

```
static inline void SPC_EnableLowPowerModeCMPBandgapBufferMode(SPC_Type *base, bool enable)
```

Enables/Disables CMP Bandgap Buffer.

This function gates CMP bandgap buffer. CMP bandgap buffer is automatically disabled and turned off in Deep Power Down mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable CMP Bandgap buffer. true - Enable Buffer Stored Reference Voltage to CMP. false - Disable Buffer Stored Reference Voltage to CMP.

```
static inline void SPC_EnableLowPowerModeCoreVDDInternalVoltageScaling(SPC_Type *base, bool enable)
```

Enables/Disables CORE VDD IVS(Internal Voltage Scaling) in low power modes.

This function gates CORE VDD IVS. When enabled, the IVS regulator will scale the external input CORE VDD to a lower voltage level to reduce internal leakage. IVS is invalid in Sleep or Deep power down mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IVS. true - enable CORE VDD IVS in Deep Sleep mode or Power Down mode. false - disable CORE VDD IVS in Deep Sleep mode or Power Down mode.

static inline void SPC_SetLowPowerWakeUpDelay(SPC_Type *base, uint16_t delay)

Sets the delay when exit the low power modes.

Parameters

- base – SPC peripheral base address.
- delay – The number of SPC timer clock cycles that the SPC waits on exit from low power modes.

status_t SPC_SetLowPowerModeRegulatorsConfig(SPC_Type *base, const
 spc_lowpower_mode_regulators_config_t
 *config)

Config regulators in Low Power mode.

This function provides the method to config all on-chip regulators in Low Power mode.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_lowpower_mode_regulators_config_t structure.

Return values

- kStatus_Success – Config regulators in Low power mode successful.
- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in Low Power mode is wrong.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level is wrong.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_SPC_CORELDOVoltageSetFail – Fail to change Core LDO voltage level.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_SPC_DCDCPulseRefreshModeIgnore – Set driver strength to Pulse Refresh mode will be ignored.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low Drive Strength will be ignored.

static inline void SPC_DisableLowPowerModeVddCoreGlitchDetect(SPC_Type *base, bool disable)

Disable/Enable VDD Core Glitch Detect in low power mode.

Note: State of glitch detect disable feature will be ignored if bandgap is disabled and glitch detect hardware will be forced to OFF state.

Parameters

- base – SPC peripheral base address.
- disable – Used to disable/enable VDD Core Glitch detect feature.
 - **true** Disable VDD Core Low Voltage detect;
 - **false** Enable VDD Core Low Voltage detect.

static inline uint8_t SPC_GetVoltageDetectStatusFlag(SPC_Type *base)

Get Voltage Detect Status Flags.

Parameters

- base – SPC peripheral base address.

Returns

Voltage Detect Status Flags. See `_spc_voltage_detect_flags` for details.

static inline void SPC_ClearVoltageDetectStatusFlag(SPC_Type *base, uint8_t mask)

Clear Voltage Detect Status Flags.

Parameters

- base – SPC peripheral base address.
- mask – The mask of the voltage detect status flags. See `_spc_voltage_detect_flags` for details.

void SPC_SetCoreVoltageDetectConfig(SPC_Type *base, const *spc_core_voltage_detect_config_t* *config)

Configs CORE voltage detect options.

This function config CORE voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to `spc_core_voltage_detect_config_t` structure.

static inline void SPC_LockCoreVoltageDetectResetSetting(SPC_Type *base)

Locks Core voltage detect reset setting.

This function locks core voltage detect reset setting. After invoking this function any configuration of Core voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockCoreVoltageDetectResetSetting(SPC_Type *base)

Unlocks Core voltage detect reset setting.

This function unlocks core voltage detect reset setting. If locks the Core voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeCoreHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core High Voltage Detector in Active mode.

Note: If the CORE_LDO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core High voltage detector in active mode. false - Disable Core High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable Core High Voltage Detect successfully.

status_t SPC_EnableActiveModeCoreLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core Low Voltage Detector in Active mode.

Note: If the CORE_LDO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core LVD. true - Enable Core Low voltage detector in active mode. false - Disable Core Low voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable Core Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeCoreHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core High Voltage Detector in Low Power mode.

This function enables/disables the Core High Voltage Detector. If enabled the Core High Voltage detector. The Bandgap mode in low power mode must be programmed so that Bandgap is enabled.

Note: If the CORE_LDO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in low power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core High voltage detector in low power mode. false - Disable Core High voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable Core High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeCoreLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core Low Voltage Detector in Low Power mode.

This function enables/disables the Core Low Voltage Detector. If enabled the Core Low Voltage detector. The Bandgap mode in low power mode must be programmed so that Bandgap is enabled.

Note: If the CORE_LDO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core Low voltage detector in low power mode. false - Disable Core Low voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable Core Low Voltage Detect in low power mode successfully.

void SPC_SetSystemVDDLowVoltageLevel(SPC_Type *base, *spc_low_voltage_level_select_t* level)
Set system VDD Low-voltage level selection.

This function selects the system VDD low-voltage level. Changing system VDD low-voltage level must be done after disabling the System VDD low voltage reset and interrupt.

Deprecated:

In latest RM, reserved for all devices, will removed in next release.

Parameters

- base – SPC peripheral base address.
- level – System VDD Low-Voltage level selection.

void SPC_SetSystemVoltageDetectConfig(SPC_Type *base, const
spc_system_voltage_detect_config_t *config)

Configs SYS voltage detect options.

This function config SYS voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to *spc_system_voltage_detect_config_t* structure.

static inline void SPC_LockSystemVoltageDetectResetSetting(SPC_Type *base)

Lock System voltage detect reset setting.

This function locks system voltage detect reset setting. After invoking this function any configuration of System Voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockSystemVoltageDetectResetSetting(SPC_Type *base)

Unlock System voltage detect reset setting.

This function unlocks system voltage detect reset setting. If locks the System voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeSystemHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System High Voltage Detector in Active mode.

Note: If the System_LDO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System High voltage detector in active mode. false - Disable System High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable System High Voltage Detect successfully.

status_t SPC_EnableActiveModeSystemLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disable the System Low Voltage Detector in Active mode.

Note: If the System_LDO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System LVD. true - Enable System Low voltage detector in active mode. false - Disable System Low voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable the System Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeSystemHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System High Voltage Detector in Low Power mode.

Note: If the System_LDO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System High voltage detector in low power mode. false - Disable System High voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable System High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeSystemLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System Low Voltage Detector in Low Power mode.

Note: If the System_LDO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System Low voltage detector in low power mode. false - Disable System Low voltage detector in low power mode.

Return values

kStatus_Success – Enables System Low Voltage Detect in low power mode successfully.

void SPC_SetIOVDDLowVoltageLevel(SPC_Type *base, *spc_low_voltage_level_select_t* level)

Set IO VDD Low-Voltage level selection.

This function selects the IO VDD Low-voltage level. Changing IO VDD low-voltage level must be done after disabling the IO VDD low voltage reset and interrupt.

Parameters

- base – SPC peripheral base address.
- level – IO VDD Low-voltage level selection.

void SPC_SetIOVoltageDetectConfig(SPC_Type *base, const *spc_io_voltage_detect_config_t* *config)

Configs IO voltage detect options.

This function config IO voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to *spc_voltage_detect_config_t* structure.

static inline void SPC_LockIOVoltageDetectResetSetting(SPC_Type *base)

Lock IO Voltage detect reset setting.

This function locks IO voltage detect reset setting. After invoking this function any configuration of system voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockIOVoltageDetectResetSetting(SPC_Type *base)

Unlock IO voltage detect reset setting.

This function unlocks IO voltage detect reset setting. If locks the IO voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeIOHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO High Voltage Detector in Active mode.

Note: If the IO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO HVD. true - Enable IO High voltage detector in active mode. false - Disable IO High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable IO High Voltage Detect successfully.

status_t SPC_EnableActiveModeIOLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO Low Voltage Detector in Active mode.

Note: If the IO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO LVD. true - Enable IO Low voltage detector in active mode. false - Disable IO Low voltage detector in active mode.

Return values

kStatus_Success – Enable IO Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeIOHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO High Voltage Detector in Low Power mode.

Note: If the IO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO HVD. true - Enable IO High voltage detector in low power mode. false - Disable IO High voltage detector in low power mode.

Return values

kStatus_Success – Enable IO High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeIOLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO Low Voltage Detector in Low Power mode.

Note: If the IO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO LVD. true - Enable IO Low voltage detector in low power mode. false - Disable IO Low voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable IO Low Voltage Detect in low power mode successfully.

void SPC_SetExternalVoltageDomainsConfig(SPC_Type *base, uint8_t lowPowerIsoMask, uint8_t IsoMask)

Configs external voltage domains.

This function configs external voltage domains isolation.

Parameters

- base – SPC peripheral base address.
- lowPowerIsoMask – The mask of external domains isolate enable during low power mode. Please read the Reference Manual for the Bitmap.
- IsoMask – The mask of external domains isolate. Please read the Reference Manual for the Bitmap.

static inline uint8_t SPC_GetExternalDomainsStatus(SPC_Type *base)

Gets External Domains status.

This function configs external voltage domains status.

Parameters

- base – SPC peripheral base address.

Returns

The status of each external domain.

static inline void SPC_EnableCoreLDORegulator(SPC_Type *base, bool enable)

Enable/Disable Core LDO regulator.

Note: The CORE LDO enable bit is write-once.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable CORE LDO Regulator. true - Enable CORE LDO Regulator. false - Disable CORE LDO Regulator.

static inline void SPC_PullDownCoreLDORegulator(SPC_Type *base, bool pulldown)
Enable/Disable the CORE LDO Regulator pull down in Deep Power Down.

Note: This function only useful when enabled the CORE LDO Regulator.

Parameters

- base – SPC peripheral base address.
- pulldown – Enable/Disable CORE LDO pulldown in Deep Power Down mode.
true - CORE LDO Regulator will discharge in Deep Power Down mode. false
- CORE LDO Regulator will not discharge in Deep Power Down mode.

status_t SPC_SetActiveModeCoreLDORegulatorConfig(SPC_Type *base, const
 spc_active_mode_core_ldo_option_t
 *option)

Configs Core LDO VDD Regulator in Active mode.

Note: If any voltage detect feature is enabled in Active mode, then CORE_LDO's drive strength must not set to low.

Note: Core VDD level for the Core LDO low power regulator can only be changed when CORELDO_VDD_DS is normal

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_active_mode_core_ldo_option_t structure.

Return values

- kStatus_Success – Config Core LDO regulator in Active power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – If any voltage detect enabled, core_ldo's drive strength can not set to low.
- kStatus_SPC_CORELDOLowVoltageWrong – The selected voltage level in active mode is not allowed.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetLowPowerModeCoreLDORegulatorConfig(SPC_Type *base, const
 spc_lowpower_mode_core_ldo_option_t
 *option)

Configs CORE LDO Regulator in low power mode.

This function configs CORE LDO Regulator in Low Power mode. If CORE LDO VDD Drive Strength is set to Normal, the CORE LDO VDD regulator voltage level in Active mode must be equal to the voltage level in Low power mode. And the Bandgap must be programmed to select bandgap enabled. Core VDD voltage levels for the Core LDO low power regulator can only be changed when the CORE LDO Drive Strength set as Normal.

Parameters

- base – SPC peripheral base address.

- `option` – Pointer to the `spc_lowpower_mode_core_ldo_option_t` structure.

Return values

- `kStatus_Success` – Config Core LDO regulator in power mode successfully.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_CORELDOLowDriveStrengthIgnore` – Set driver strength to low will be ignored.
- `kStatus_SPC_CORELDIVoltageSetFail` – Fail to change Core LDO voltage level.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

static inline void SPC_EnableSystemLDORegulator(SPC_Type *base, bool enable)
Enable/Disable System LDO regulator.

Note: The SYSTEM LDO enable bit is write-once.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Enable/Disable System LDO Regulator. true - Enable System LDO Regulator. false - Disable System LDO Regulator.

static inline void SPC_EnableSystemLDOSinkFeature(SPC_Type *base, bool sink)
Enable/Disable current sink feature of System LDO Regulator.

Parameters

- `base` – SPC peripheral base address.
- `sink` – Enable/Disable current sink feature. true - Enable current sink feature of System LDO Regulator. false - Disable current sink feature of System LDO Regulator.

status_t SPC_SetActiveModeSystemLDORegulatorConfig(SPC_Type *base, const
spc_active_mode_sys_ldo_option_t
*option)

Configs System LDO VDD Regulator in Active mode.

This function configs System LDO VDD Regulator in Active mode. If System LDO VDD Drive Strength is set to Normal, the Bandgap mode in Active mode must be programmed to a value that enables the bandgap. If any voltage detects are kept enabled, configuration to set System LDO VDD drive strength to low will be ignored. If select System LDO VDD Regulator voltage level to Over Drive Voltage, the Drive Strength of System LDO VDD Regulator must be set to Normal otherwise the regulator Drive Strength will be forced to Normal. If select System LDO VDD Regulator voltage level to Over Drive Voltage, the High voltage detect must be disabled. Otherwise it will be fail to regulator to Over Drive Voltage.

Parameters

- `base` – SPC peripheral base address.
- `option` – Pointer to the `spc_active_mode_sys_ldo_option_t` structure.

Return values

- `kStatus_Success` – Config System LDO regulator in Active power mode successful.

- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_SYSLDOOverDriveVoltageFail` – Fail to regulator to Over Drive Voltage.
- `kStatus_SPC_SYSLDOLowDriveStrengthIgnore` – Set driver strength to Low will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`status_t` SPC_SetLowPowerModeSystemLDORegulatorConfig(SPC_Type *base, const *spc_lowpower_mode_sys_ldo_option_t* *option)

Configs System LDO regulator in low power modes.

This function configs System LDO regulator in low power modes. If System LDO VDD Regulator Drive strength is set to normal, bandgap mode in low power mode must be programmed to a value that enables the Bandgap. If any High voltage detectors or Low Voltage detectors are kept enabled, configuration to set System LDO Regulator drive strength as Low will be ignored.

Parameters

- `base` – SPC peripheral base address.
- `option` – Pointer to `spc_lowpower_mode_sys_ldo_option_t` structure.

Return values

- `kStatus_Success` – Config System LDO regulator in Low Power Mode successfully.
- `kStatus_SPC_Busy` – The SPC instance is busy to execute any type of power mode transition.
- `kStatus_SPC_SYSLDOLowDriveStrengthIgnore` – Set driver strength to low will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

static inline void SPC_EnableDCDCRegulator(SPC_Type *base, bool enable)
Enable/Disable DCDC Regulator.

Note: The DCDC enable bit is write-once.

Parameters

- `base` – SPC peripheral base address.
- `enable` – Enable/Disable DCDC Regulator. `true` - Enable DCDC Regulator. `false` - Disable DCDC Regulator.

`status_t` SPC_SetDCDCBurstConfig(SPC_Type *base, *spc_dcdc_burst_config_t* *config)
Config DCDC Burst options.

Parameters

- `base` – SPC peripheral base address.
- `config` – Pointer to `spc_dcdc_burst_config_t` structure.

Return values

- `kStatus_Success` – Successfully configured.
- `kStatus_Timeout` – Timeout occurs while waiting completion.


```
void SPC_SetDCDCRefreshCount(SPC_Type *base, uint16_t count)
```

Set the count value of the reference clock.

This function set the count value of the reference clock to control the frequency of dcdc refresh when dcdc is configured in Pulse Refresh mode.

Parameters

- base – SPC peripheral base address.
- count – The count value, 16 bit width.

```
status_t SPC_SetActiveModeDCDCRegulatorConfig(SPC_Type *base, const
                                              spc_active_mode_dcdc_option_t *option)
```

Configs DCDC VDD Regulator in Active mode.

This function configs DCDC VDD Regulator in Active mode. If DCDC VDD Drive Strength is set to Normal, the Bandgap mode in Active mode must be programmed to a value that enable the bandgap. If any voltage detectors are kept enabled, configuration to set DCDC VDD drive strength to low will be ignored. When switching DCDC from low drive strength to Normal driver strength, make sure the DCDC high VDD LVL setting to the same level that was set prior to switching the DCDC to low drive strength.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_active_mode_dcdc_option_t structure.

Return values

- kStatus_Success – Config DCDC regulator in Active power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
status_t SPC_SetLowPowerModeDCDCRegulatorConfig(SPC_Type *base, const
                                              spc_lowpower_mode_dcdc_option_t
                                              *option)
```

Configs DCDC VDD Regulator in Low power modes.

This function configs DCDC VDD Regulator in Low Power modes. If DCDC VDD Drive Strength is set to Normal, the Bandgap mode in Low Power mode must be programmed to a value that enables the Bandgap. If any of voltage detectors are kept enabled, configuration to set DCDC VDD Drive Strength to Low or Pulse mode will be ignored. In Deep Power Down mode, DCDC regulator is always turned off.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_lowpower_mode_dcdc_option_t structure.

Return values

- kStatus_Success – Config DCDC regulator in low power mode successfully.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_DCDCPulseRefreshModeIgnore – Set driver strength to Pulse Refresh mode will be ignored.

- `kStatus_SPC_DCDCLowDriveStrengthIgnore` – Set driver strength to Low Drive Strength will be ignored.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`status_t SPC_SetSRAMOperateVoltage(SPC_Type *base, spc_sram_operat_voltage_t voltage)`
Set the SRAM operate voltage level.

Parameters

- `base` – SPC peripheral base address.
- `voltage` – Target SRAM operate voltage level, please refer to `spc_sram_operat_voltage_t`.

Return values

- `kStatus_Success` – Successfully configured.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`void SPC_ConfigVddCoreGlitchDetector(SPC_Type *base, const spc_vdd_core_glitch_detector_config_t *config)`

Configures VDD Core Glitch detector, including ripple counter selection, timeout value and so on.

Parameters

- `base` – SPC peripheral base address.
- `config` – Pointer to the structure in type of `spc_vdd_core_glitch_detector_config_t`.

`static inline bool SPC_CheckGlitchRippleCounterOutput(SPC_Type *base, spc_vdd_core_glitch_ripple_counter_select_t rippleCounter)`

Checks selected 4-bit glitch ripple counter's output.

Parameters

- `base` – SPC peripheral base address.
- `rippleCounter` – The ripple counter to check, please refer to `spc_vdd_core_glitch_ripple_counter_select_t`.

Return values

- `true` – The selected ripple counter output is 1, will generate interrupt or reset based on settings.
- `false` – The selected ripple counter output is 0.

`static inline void SPC_ClearGlitchRippleCounterOutput(SPC_Type *base, spc_vdd_core_glitch_ripple_counter_select_t rippleCounter)`

Clears output of selected glitch ripple counter.

Parameters

- `base` – SPC peripheral base address.
- `rippleCounter` – The ripple counter to check, please refer to `spc_vdd_core_glitch_ripple_counter_select_t`.

`static inline void SPC_LockVddCoreVoltageGlitchDetectResetControl(SPC_Type *base)`

After invoking this function, writes to `SPC_VDD_CORE_GLITCH_DETECT_SC[RE]` register are ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockVddCoreVoltageGlitchDetectResetControl(SPC_Type *base)

After invoking this function, writes to SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register are allowed.

Parameters

- base – SPC peripheral base address.

static inline bool SPC_CheckVddCoreVoltageGlitchResetControlState(SPC_Type *base)

Checks if SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is writable.

Parameters

- base – SPC peripheral base address.

Return values

- true – SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is writable.
- false – SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is not writable.

FSL_SPC_DRIVER_VERSION

SPC driver version 2.8.0.

SPC_EVD_CFG_REG_EVDISO_SHIFT

SPC_EVD_CFG_REG_EVDLPISO_SHIFT

SPC_EVD_CFG_REG_EVDSTAT_SHIFT

SPC_EVD_CFG_REG_EVDISO(x)

SPC_EVD_CFG_REG_EVDLPISO(x)

SPC_EVD_CFG_REG_EVDSTAT(x)

struct _spc_lowpower_request_config

#include <fsl_spc.h> Low Power Request output pin configuration.

Public Members

bool enable

Low Power Request Output enable.

spc_lowpower_request_pin_polarity_t polarity

Low Power Request Output pin polarity select.

spc_lowpower_request_output_override_t override

Low Power Request Output Override.

struct _spc_integrated_power_switch_config

#include <fsl_spc.h> Integrated power switch configuration.

Note: Legacy structure, will be removed.

Public Members

bool wakeup

Assert an output pin to un-gate the integrated power switch.

bool sleep

Assert an output pin to power gate the intergrated power switch.

struct _spc_active_mode_core_ldo_option

#include <fsl_spc.h> Core LDO regulator options in Active mode.

Public Members

spc_core_ldo_voltage_level_t CoreLDOVoltage

Core LDO Regulator Voltage Level selection in Active mode.

spc_core_ldo_drive_strength_t CoreLDODriveStrength

Core LDO Regulator Drive Strength selection in Active mode

struct _spc_active_mode_sys_ldo_option

#include <fsl_spc.h> System LDO regulator options in Active mode.

Public Members

spc_sys_ldo_voltage_level_t SysLDOVoltage

System LDO Regulator Voltage Level selection in Active mode.

spc_sys_ldo_drive_strength_t SysLDODriveStrength

System LDO Regulator Drive Strength selection in Active mode.

struct _spc_active_mode_dcdc_option

#include <fsl_spc.h> DCDC regulator options in Active mode.

Public Members

spc_dcdc_voltage_level_t DCDCVoltage

DCDC Regulator Voltage Level selection in Active mode.

spc_dcdc_drive_strength_t DCDCDriveStrength

DCDC VDD Regulator Drive Strength selection in Active mode.

struct _spc_lowpower_mode_core_ldo_option

#include <fsl_spc.h> Core LDO regulator options in Low Power mode.

Public Members

spc_core_ldo_voltage_level_t CoreLDOVoltage

Core LDO Regulator Voltage Level selection in Low Power mode.

spc_core_ldo_drive_strength_t CoreLDODriveStrength

Core LDO Regulator Drive Strength selection in Low Power mode

struct _spc_lowpower_mode_sys_ldo_option

#include <fsl_spc.h> System LDO regulator options in Low Power mode.

Public Members

spc_sys_ldo_drive_strength_t SysLDODriveStrength

System LDO Regulator Drive Strength selection in Low Power mode.

struct _spc_lowpower_mode_dcdc_option

#include <fsl_spc.h> DCDC regulator options in Low Power mode.

Public Members

spc_dcdc_voltage_level_t DCDCVoltage

DCDC Regulator Voltage Level selection in Low Power mode.

spc_dcdc_drive_strength_t DCDCDriveStrength

DCDC VDD Regulator Drive Strength selection in Low Power mode.

struct *_spc_voltage_detect_option*

#include <fsl_spc.h> CORE/SYS/IO VDD Voltage Detect options.

Public Members

bool HVDInterruptEnable

CORE/SYS/IO VDD High Voltage Detect interrupt enable.

bool HVDRResetEnable

CORE/SYS/IO VDD High Voltage Detect reset enable.

bool LVDInterruptEnable

CORE/SYS/IO VDD Low Voltage Detect interrupt enable.

bool LVDRResetEnable

CORE/SYS/IO VDD Low Voltage Detect reset enable.

struct *_spc_dcdc_burst_config*

#include <fsl_spc.h> DCDC Burst configuration.

Public Members

bool softwareBurstRequest

Enable/Disable DCDC Software Burst Request.

bool externalBurstRequest

Enable/Disable DCDC External Burst Request.

bool stabilizeBurstFreq

Enable/Disable DCDC frequency stabilization.

uint8_t freq

The frequency of the current burst.

struct *_spc_core_voltage_detect_config*

#include <fsl_spc.h> Core Voltage Detect configuration.

Public Members

spc_voltage_detect_option_t option

Core VDD Voltage Detect option.

struct *_spc_system_voltage_detect_config*

#include <fsl_spc.h> System Voltage Detect Configuration.

Public Members

spc_voltage_detect_option_t option

System VDD Voltage Detect option.

spc_low_voltage_level_select_t level

Deprecated:

, reserved for all devices, will removed in next release.

struct *_spc_io_voltage_detect_config*

#include <fsl_spc.h> IO Voltage Detect Configuration.

Public Members

spc_voltage_detect_option_t option

IO VDD Voltage Detect option.

spc_low_voltage_level_select_t level

IO VDD Low-voltage level selection.

struct *_spc_active_mode_regulators_config*

#include <fsl_spc.h> Active mode configuration.

struct *_spc_lowpower_mode_regulators_config*

#include <fsl_spc.h> Low Power Mode configuration.

struct *_spc_vdd_core_glitch_detector_config*

#include <fsl_spc.h> The configuration of VDD Core glitch detector.

Public Members

spc_vdd_core_glitch_ripple_counter_select_t rippleCounterSelect

Used to set ripple counter.

uint8_t resetTimeoutValue

The timeout value used to reset glitch detect/compare logic after an initial glitch is detected.

bool enableReset

Used to enable/disable POR/LVD reset that caused by CORE VDD glitch detect error.

bool enableInterrupt

Used to enable/disable hardware interrupt if CORE VDD glitch detect error.

2.68 SYSPM: System Performance Monitor

enum *_syspm_monitor*

syspm select control monitor

Values:

enumerator *kSYSPM_Monitor0*

Monitor 0

enum *_syspm_event*

syspm select event

Values:

enumerator *kSYSPM_Event1*

Event 1

```
enumerator kSYSPM_Event2
    Event 2
enumerator kSYSPM_Event3
    Event 3
enum __syspm_mode
    syspm set count mode
Values:
enumerator kSYSPM_BothMode
    count in both modes
enumerator kSYSPM_UserMode
    count only in user mode
enumerator kSYSPM_PrivilegedMode
    count only in privileged mode
enum __syspm_startstop_control
    syspm start/stop control
Values:
enumerator kSYSPM_Idle
    idle >
enumerator kSYSPM_LocalStop
    local stop
enumerator kSYSPM_LocalStart
    local start
enumerator kSYSPM_EnableTraceControl
    enable global TSTART/TSTOP
enumerator kSYSPM_GlobalStart
    global stop
enumerator kSYSPM_GlobalStop
    global start
typedef enum __syspm_monitor syspm_monitor_t
    syspm select control monitor
typedef enum __syspm_event syspm_event_t
    syspm select event
typedef enum __syspm_mode syspm_mode_t
    syspm set count mode
typedef enum __syspm_startstop_control syspm_startstop_control_t
    syspm start/stop control
void SYSPM_Init(SYSPM_Type *base)
    Initializes the SYSPM.
    This function enables the SYSPM clock.
```

Parameters

- base – SYSPM peripheral base address.

void SYSPM_Deinit(SYSPM_Type *base)

Deinitializes the SYSPM.

This function disables the SYSPM clock.

Parameters

- base – SYSPM peripheral base address.

void SYSPM_SelectEvent(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event, uint8_t eventCode)

Select event counters.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.
- event – syspm select event, see to *syspm_event_t*.
- eventCode – select which event to be counted in PMECTR_x, see to table Events.

void SYSPM_ResetEvent(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event)

Reset event counters.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.
- event – syspm select event, see to *syspm_event_t*.

void SYSPM_ResetInstructionEvent(SYSPM_Type *base, *syspm_monitor_t* monitor)

Reset Instruction Counter.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.

void SYSPM_SetCountMode(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_mode_t* mode)

Set count mode.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.
- mode – syspm select counter mode, see to *syspm_mode_t*.

void SYSPM_SetStartStopControl(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_startstop_control_t* ssc)

Set Start/Stop Control.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.
- ssc – This 3-bit field provides a three-phase mechanism to start/stop the counters. It includes a prioritized scheme with local start > local stop > global start > global stop > conditional TSTART > TSTOP. The global and conditional start/stop affect all configured PM/PSAM module concurrently so counters are “coherent”. see to *syspm_startstop_control_t*

void SYSPM_DisableCounter(SYSPM_Type *base, *syspm_monitor_t* monitor)

Disable Counters if Stopped or Halted.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.

uint64_t SYSPM_GetEventCounter(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event)

This is the the 40-bits of eventx counter. The value in this register increments each time the event selected in PMCRx[SELEVTx] occurs.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to *syspm_monitor_t*.
- event – syspm select event, see to *syspm_event_t*.

Returns

- When the return value is not equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents a 40 bits eventx counter.
- When the return value is equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents timeout occurred.

EVENT_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM event count stable (0 means wait forever)

INSTRUCTION_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM instruction count stable (0 means wait forever)

FSL_SYSPM_DRIVER_VERSION

SYSPM driver version.

SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE

2.69 TPM: Timer PWM Module

uint32_t TPM_GetInstance(TPM_Type *base)

Gets the instance from the base address.

Parameters

- base – TPM peripheral base address

Returns

The TPM instance

void TPM_Init(TPM_Type *base, const *tpm_config_t* *config)

Ungates the TPM clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the TPM driver.

Parameters

- base – TPM peripheral base address

- config – Pointer to user's TPM config structure.

void TPM_Deinit(TPM_Type *base)

Stops the counter and gates the TPM clock.

Parameters

- base – TPM peripheral base address

void TPM_GetDefaultConfig(*tpm_config_t* *config)

Fill in the TPM config struct with the default settings.

The default values are:

```
config->prescale = kTPM_Prescale_Divide_1;
config->useGlobalTimeBase = false;
config->syncGlobalTimeBase = false;
config->dozeEnable = false;
config->dbgMode = false;
config->enableReloadOnTrigger = false;
config->enableStopOnOverflow = false;
config->enableStartOnTrigger = false;
#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
config->enablePauseOnTrigger = false;
#endif
config->triggerSelect = kTPM_Trigger_Select_0;
#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
config->triggerSource = kTPM_TriggerSource_External;
config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
#endif
#if defined(FSL_FEATURE_TPM_HAS_POL) && FSL_FEATURE_TPM_HAS_POL
config->chnlPolarity = 0U;
#endif
```

Parameters

- config – Pointer to user's TPM config structure.

tpm_clock_prescale_t TPM_CalculateCounterClkDiv(TPM_Type *base, uint32_t
counterPeriod_Hz, uint32_t srcClock_Hz)

Calculates the counter clock prescaler.

This function calculates the values for SC[PS].

return Calculated clock prescaler value.

Parameters

- base – TPM peripheral base address
- counterPeriod_Hz – The desired frequency in Hz which corresponding to the time when the counter reaches the mod value
- srcClock_Hz – TPM counter clock in Hz

static inline void TPM_Reset(TPM_Type *base)

Performs a software reset on the TPM module.

Reset all internal logic and registers, except the Global Register. Remains set until cleared by software.

Note: TPM software reset is available on certain SoC's only

Parameters

- base – TPM peripheral base address

status_t TPM_SetupPwm(TPM_Type *base, const *tpm_chnl_pwm_signal_param_t* *chnlParams, uint8_t numOfChnls, *tpm_pwm_mode_t* mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)

Configures the PWM signal parameters.

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

- base – TPM peripheral base address
- chnlParams – Array of PWM channel parameters to configure the channel(s)
- numOfChnls – Number of channels to configure, this should be the size of the array passed in
- mode – PWM operation mode, options available in enumeration *tpm_pwm_mode_t*
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – TPM counter clock in Hz

Returns

kStatus_Success PWM setup successful kStatus_Error PWM setup failed kStatus_Timeout PWM setup timeout when write register CnV or MOD

status_t TPM_UpdatePwmDutycycle(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_pwm_mode_t* currentPwmMode, uint8_t dutyCyclePercent)

Update the duty cycle of an active PWM signal.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number. In combined mode, this represents the channel pair number
- currentPwmMode – The current PWM mode set during PWM setup
- dutyCyclePercent – New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

void TPM_UpdateChnlEdgeLevelSelect(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t level)

Update the edge level selection for a channel.

Note: When the TPM has PWM pause level select feature (FSL_FEATURE_TPM_HAS_PAUSE_LEVEL_SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use TPM_DisableChannel API to close the PWM output.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

- level – The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field.

static inline uint8_t TPM_GetChannelControlBits(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Get the channel control bits value (mode, edge and level bit fields).

This function disables the channel by clearing all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The control bits value. This is the logical OR of members of the enumeration *tpm_chnl_control_bit_mask_t*.

static inline *status_t* TPM_DisableChannel(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Disable the channel.

This function disables the channel by clearing all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout when writing register CnSC

static inline *status_t* TPM_EnableChannel(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t control)

Enable the channel according to mode and level configs.

This function enables the channel output according to input mode/level config parameters.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- control – The control bits value. This is the logical OR of members of the enumeration *tpm_chnl_control_bit_mask_t*.

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout when writing register CnSC

void TPM_SetupInputCapture(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_input_capture_edge_t* captureMode)

Enables capturing an input signal on the channel using the function parameters.

When the edge specified in the *captureMode* argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- captureMode – Specifies which edge to capture

```
status_t TPM_SetupOutputCompare(TPM_Type *base, tpm_chnl_t chnlNumber,  
                                tpm_output_compare_mode_t compareMode, uint32_t  
                                compareValue)
```

Configures the TPM to generate timed pulses.

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- compareMode – Action to take on the channel output when the compare condition is met
- compareValue – Value to be programmed in the CnV register.

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout when write register CnV

```
void TPM_SetupDualEdgeCapture(TPM_Type *base, tpm_chnl_t chnlPairNumber, const  
                             tpm_dual_edge_capture_param_t *edgeParam, uint32_t  
                             filterValue)
```

Configures the dual edge capture mode of the TPM.

This function allows to measure a pulse width of the signal on the input of channel of a channel pair. The filter function is disabled if the filterVal argument passed is zero.

Parameters

- base – TPM peripheral base address
- chnlPairNumber – The TPM channel pair number; options are 0, 1, 2, 3
- edgeParam – Sets up the dual edge capture function
- filterValue – Filter value, specify 0 to disable filter.

```
void TPM_SetupQuadDecode(TPM_Type *base, const tpm_phase_params_t *phaseAParams,  
                        const tpm_phase_params_t *phaseBParams,  
                        tpm_quad_decode_mode_t quadMode)
```

Configures the parameters and activates the quadrature decode mode.

Parameters

- base – TPM peripheral base address
- phaseAParams – Phase A configuration parameters
- phaseBParams – Phase B configuration parameters
- quadMode – Selects encoding mode used in quadrature decoder mode

```
static inline void TPM_SetChannelPolarity(TPM_Type *base, tpm_chnl_t chnlNumber, bool  
                                         enable)
```

Set the input and output polarity of each of the channels.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- enable – true: Set the channel polarity to active high; false: Set the channel polarity to active low;

```
static inline void TPM_EnableChannelExtTrigger(TPM_Type *base, tpm_chnl_t chnlNumber, bool enable)
```

Enable external trigger input to be used by channel.

In input capture mode, configures the trigger input that is used by the channel to capture the counter value. In output compare or PWM mode, configures the trigger input used to modulate the channel output. When modulating the output, the output is forced to the channel initial value whenever the trigger is not asserted.

Note: No matter how many external trigger sources there are, only input trigger 0 and 1 are used. The even numbered channels share the input trigger 0 and the odd numbered channels share the second input trigger 1.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- enable – true: Configures trigger input 0 or 1 to be used by channel; false: Trigger input has no effect on the channel

```
void TPM_EnableInterrupts(TPM_Type *base, uint32_t mask)
```

Enables the selected TPM interrupts.

Parameters

- base – TPM peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `tpm_interrupt_enable_t`

```
void TPM_DisableInterrupts(TPM_Type *base, uint32_t mask)
```

Disables the selected TPM interrupts.

Parameters

- base – TPM peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `tpm_interrupt_enable_t`

```
uint32_t TPM_GetEnabledInterrupts(TPM_Type *base)
```

Gets the enabled TPM interrupts.

Parameters

- base – TPM peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `tpm_interrupt_enable_t`

```
void TPM_RegisterCallBack(TPM_Type *base, tpm_callback_t callback)
```

Register callback.

If channel or overflow interrupt is enabled by the user, then a callback can be registered which will be invoked when the interrupt is triggered.

Parameters

- base – TPM peripheral base address
- callback – Callback function

void TPM_DriverIRQHandler(uint32_t instance)

TPM driver IRQ handler common entry.

This function provides the common IRQ request entry for TPM.

Parameters

- instance – TPM instance.

static inline uint32_t TPM_GetChannelValue(TPM_Type *base, tpm_chnl_t chnlNumber)

Gets the TPM channel value.

Note: The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The channle CnV regisyer value.

static inline uint32_t TPM_GetStatusFlags(TPM_Type *base)

Gets the TPM status flags.

Parameters

- base – TPM peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration tpm_status_flags_t

static inline void TPM_ClearStatusFlags(TPM_Type *base, uint32_t mask)

Clears the TPM status flags.

Parameters

- base – TPM peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration tpm_status_flags_t

static inline status_t TPM_SetTimerPeriod(TPM_Type *base, uint32_t ticks)

Sets the timer period in units of ticks.

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note:

- This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.
 - Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.
-

Parameters

- base – TPM peripheral base address
- ticks – A timer period in units of ticks, which should be equal or greater than 1.

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

static inline uint32_t TPM_GetCurrentTimerCount(TPM_Type *base)

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

- base – TPM peripheral base address

Returns

The current counter value in ticks

static inline void TPM_StartTimer(TPM_Type *base, tpm_clock_source_t clockSource)

Starts the TPM counter.

Parameters

- base – TPM peripheral base address
- clockSource – TPM clock source; once clock source is set the counter will start running

static inline status_t TPM_StopTimer(TPM_Type *base)

Stops the TPM counter.

Parameters

- base – TPM peripheral base address

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

FSL_TPM_DRIVER_VERSION

TPM driver version 2.4.0.

enum _tpm_chnl

List of TPM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kTPM_Chnl_0
TPM channel number 0

enumerator kTPM_Chnl_1
TPM channel number 1

enumerator kTPM_Chnl_2
TPM channel number 2

enumerator kTPM_Chnl_3
TPM channel number 3

enumerator kTPM_Chnl_4
TPM channel number 4

enumerator kTPM_Chnl_5
TPM channel number 5

enumerator kTPM_Chnl_6
TPM channel number 6

enumerator kTPM_Chnl_7
TPM channel number 7

enum _tpm_pwm_mode
TPM PWM operation modes.

Values:

enumerator kTPM_EdgeAlignedPwm
Edge aligned PWM

enumerator kTPM_CenterAlignedPwm
Center aligned PWM

enumerator kTPM_CombinedPwm
Combined PWM (Edge-aligned, center-aligned, or asymmetrical PWMs can be obtained in combined mode using different software configurations)

enum _tpm_pwm_level_select
TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Values:

enumerator kTPM_HighTrue
High true pulses

enumerator kTPM_LowTrue
Low true pulses

enum _tpm_pwm_pause_level_select
TPM PWM output when first enabled or paused: set or clear.

Values:

enumerator kTPM_ClearOnPause
Clear Output when counter first enabled or paused.

enumerator kTPM_SetOnPause
Set Output when counter first enabled or paused.

enum _tpm_chnl_control_bit_mask
List of TPM channel modes and level control bit mask.

Values:

enumerator kTPM_ChnlELSnAMask
Channel ELSA bit mask.

enumerator kTPM_ChnlELSnBMask

Channel ELSB bit mask.

enumerator kTPM_ChnlMSAMask

Channel MSA bit mask.

enumerator kTPM_ChnlMSBMask

Channel MSB bit mask.

enum _tpm_trigger_select

Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

Values:

enumerator kTPM_Trigger_Select_0

enumerator kTPM_Trigger_Select_1

enumerator kTPM_Trigger_Select_2

enumerator kTPM_Trigger_Select_3

enumerator kTPM_Trigger_Select_4

enumerator kTPM_Trigger_Select_5

enumerator kTPM_Trigger_Select_6

enumerator kTPM_Trigger_Select_7

enumerator kTPM_Trigger_Select_8

enumerator kTPM_Trigger_Select_9

enumerator kTPM_Trigger_Select_10

enumerator kTPM_Trigger_Select_11

enumerator kTPM_Trigger_Select_12

enumerator kTPM_Trigger_Select_13

enumerator kTPM_Trigger_Select_14

enumerator kTPM_Trigger_Select_15

enum _tpm_trigger_source

Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

Values:

enumerator kTPM_TriggerSource_External

Use external trigger input

enumerator kTPM_TriggerSource_Internal
 Use internal trigger (channel pin input capture)

enum _tpm_ext_trigger_polarity
 External trigger source polarity.

Note: Selects the polarity of the external trigger source.

Values:

enumerator kTPM_ExtTrigger_Active_High
 External trigger input is active high

enumerator kTPM_ExtTrigger_Active_Low
 External trigger input is active low

enum _tpm_output_compare_mode
 TPM output compare modes.

Values:

enumerator kTPM_NoOutputSignal
 No channel output when counter reaches CnV

enumerator kTPM_ToggleOnMatch
 Toggle output

enumerator kTPM_ClearOnMatch
 Clear output

enumerator kTPM_SetOnMatch
 Set output

enumerator kTPM_HighPulseOutput
 Pulse output high

enumerator kTPM_LowPulseOutput
 Pulse output low

enum _tpm_input_capture_edge
 TPM input capture edge.

Values:

enumerator kTPM_RisingEdge
 Capture on rising edge only

enumerator kTPM_FallingEdge
 Capture on falling edge only

enumerator kTPM_RiseAndFallEdge
 Capture on rising or falling edge

enum _tpm_quad_decode_mode
 TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

Values:

enumerator kTPM_QuadPhaseEncode
Phase A and Phase B encoding mode

enumerator kTPM_QuadCountAndDir
Count and direction encoding mode

enum _tpm_phase_polarity
TPM quadrature phase polarities.

Values:

enumerator kTPM_QuadPhaseNormal
Phase input signal is not inverted

enumerator kTPM_QuadPhaseInvert
Phase input signal is inverted

enum _tpm_clock_source
TPM clock source selection.

Values:

enumerator kTPM_SystemClock
System clock

enumerator kTPM_ExternalClock
External TPM_EXTCLK pin clock

enumerator kTPM_ExternalInputTriggerClock
Selected external input trigger clock

enum _tpm_clock_prescale
TPM prescale value selection for the clock source.

Values:

enumerator kTPM_Prescale_Divide_1
Divide by 1

enumerator kTPM_Prescale_Divide_2
Divide by 2

enumerator kTPM_Prescale_Divide_4
Divide by 4

enumerator kTPM_Prescale_Divide_8
Divide by 8

enumerator kTPM_Prescale_Divide_16
Divide by 16

enumerator kTPM_Prescale_Divide_32
Divide by 32

enumerator kTPM_Prescale_Divide_64
Divide by 64

enumerator kTPM_Prescale_Divide_128
Divide by 128

enum _tpm_interrupt_enable
List of TPM interrupts.

Values:

enumerator kTPM_Chnl0InterruptEnable
Channel 0 interrupt.

enumerator kTPM_Chnl1InterruptEnable
Channel 1 interrupt.

enumerator kTPM_Chnl2InterruptEnable
Channel 2 interrupt.

enumerator kTPM_Chnl3InterruptEnable
Channel 3 interrupt.

enumerator kTPM_Chnl4InterruptEnable
Channel 4 interrupt.

enumerator kTPM_Chnl5InterruptEnable
Channel 5 interrupt.

enumerator kTPM_Chnl6InterruptEnable
Channel 6 interrupt.

enumerator kTPM_Chnl7InterruptEnable
Channel 7 interrupt.

enumerator kTPM_TimeOverflowInterruptEnable
Time overflow interrupt.

enum _tpm_status_flags
List of TPM flags.

Values:

enumerator kTPM_Chnl0Flag
Channel 0 flag

enumerator kTPM_Chnl1Flag
Channel 1 flag

enumerator kTPM_Chnl2Flag
Channel 2 flag

enumerator kTPM_Chnl3Flag
Channel 3 flag

enumerator kTPM_Chnl4Flag
Channel 4 flag

enumerator kTPM_Chnl5Flag
Channel 5 flag

enumerator kTPM_Chnl6Flag
Channel 6 flag

enumerator kTPM_Chnl7Flag
Channel 7 flag

enumerator kTPM_TimeOverflowFlag
Time overflow flag

typedef enum _tpm_chnl tpm_chnl_t
List of TPM channels.

Note: Actual number of available channels is SoC dependent

typedef enum *_tpm_pwm_mode* tpm_pwm_mode_t

TPM PWM operation modes.

typedef enum *_tpm_pwm_level_select* tpm_pwm_level_select_t

TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

typedef enum *_tpm_pwm_pause_level_select* tpm_pwm_pause_level_select_t

TPM PWM output when first enabled or paused: set or clear.

typedef enum *_tpm_chnl_control_bit_mask* tpm_chnl_control_bit_mask_t

List of TPM channel modes and level control bit mask.

typedef struct *_tpm_chnl_pwm_signal_param* tpm_chnl_pwm_signal_param_t

Options to configure a TPM channel's PWM signal.

typedef enum *_tpm_trigger_select* tpm_trigger_select_t

Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

typedef enum *_tpm_trigger_source* tpm_trigger_source_t

Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

typedef enum *_tpm_ext_trigger_polarity* tpm_ext_trigger_polarity_t

External trigger source polarity.

Note: Selects the polarity of the external trigger source.

typedef enum *_tpm_output_compare_mode* tpm_output_compare_mode_t

TPM output compare modes.

typedef enum *_tpm_input_capture_edge* tpm_input_capture_edge_t

TPM input capture edge.

typedef struct *_tpm_dual_edge_capture_param* tpm_dual_edge_capture_param_t

TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

typedef enum *_tpm_quad_decode_mode* tpm_quad_decode_mode_t

TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

typedef enum *_tpm_phase_polarity* tpm_phase_polarity_t

TPM quadrature phase polarities.

typedef struct *_tpm_phase_param* tpm_phase_params_t

TPM quadrature decode phase parameters.

typedef enum *_tpm_clock_source* tpm_clock_source_t

TPM clock source selection.

typedef enum *_tpm_clock_prescale* tpm_clock_prescale_t

TPM prescale value selection for the clock source.

typedef struct *_tpm_config* tpm_config_t

TPM config structure.

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the TPM_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

typedef enum *_tpm_interrupt_enable* tpm_interrupt_enable_t

List of TPM interrupts.

typedef enum *_tpm_status_flags* tpm_status_flags_t

List of TPM flags.

typedef void (*tpm_callback_t)(TPM_Type *base)

TPM callback function pointer.

Param base

TPM peripheral base address.

TPM_TIMEOUT

Max loops to wait for writing register.

When writing MOD CnV CnSC and SC register, driver will wait until register is updated. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

TPM_MAX_COUNTER_VALUE(x)

Help macro to get the max counter value.

struct *_tpm_chnl_pwm_signal_param*

#include <fsl_tpm.h> Options to configure a TPM channel's PWM signal.

Public Members

tpm_chnl_t chnlNumber

TPM channel to configure. In combined mode (available in some SoC's), this represents the channel pair number

tpm_pwm_pause_level_select_t pauseLevel

PWM output level when counter first enabled or paused

tpm_pwm_level_select_t level

PWM output active level select

uint8_t dutyCyclePercent

PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=always active signal (100% duty cycle)

uint8_t firstEdgeDelayPercent

Used only in combined PWM mode to generate asymmetrical PWM. Specifies the delay to the first edge in a PWM period. If unsure, leave as 0. Should be specified as percentage of the PWM period, (dutyCyclePercent + firstEdgeDelayPercent) value should be not greater than 100.

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

tpm_pwm_pause_level_select_t secPauseLevel

Used only in combined PWM mode. Define the second channel output level when counter first enabled or paused

uint8_t deadTimeValue[2]

The dead time value for channel n and n+1 in combined complementary PWM mode. Deadtime insertion is disabled when this value is zero, otherwise deadtime insertion for channel n/n+1 is configured as (deadTimeValue * 4) clock cycles. deadTimeValue's available range is 0 ~ 15.

struct __tpm_dual_edge_capture_param

#include <fsl_tpm.h> TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

Public Members

bool enableSwap

true: Use channel n+1 input, channel n input is ignored; false: Use channel n input, channel n+1 input is ignored

tpm_input_capture_edge_t currChanEdgeMode

Input capture edge select for channel n

tpm_input_capture_edge_t nextChanEdgeMode

Input capture edge select for channel n+1

struct __tpm_phase_param

#include <fsl_tpm.h> TPM quadrature decode phase parameters.

Public Members

uint32_t phaseFilterVal

Filter value, filter is disabled when the value is zero

tpm_phase_polarity_t phasePolarity

Phase polarity

struct __tpm_config

#include <fsl_tpm.h> TPM config structure.

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the TPM_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

tpm_clock_prescale_t prescale

Select TPM clock prescale value

bool useGlobalTimeBase

true: The TPM channels use an external global time base (the local counter still use for generate overflow interrupt and DMA request); false: All TPM channels use the local counter as their timebase

bool syncGlobalTimeBase

true: The TPM counter is synchronized to the global time base; false: disabled

tpm_trigger_select_t triggerSelect

Input trigger to use for controlling the counter operation

tpm_trigger_source_t triggerSource

Decides if we use external or internal trigger.

tpm_ext_trigger_polarity_t extTriggerPolarity

when using external trigger source, need selects the polarity of it.

bool enableDoze

true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode

bool enableDebugMode

true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode

bool enableReloadOnTrigger

true: TPM counter is reloaded on trigger; false: TPM counter not reloaded

bool enableStopOnOverflow

true: TPM counter stops after overflow; false: TPM counter continues running after overflow

bool enableStartOnTrigger

true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately

bool enablePauseOnTrigger

true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running

uint8_t chnlPolarity

Defines the input/output polarity of the channels in POL register

2.70 TRDC: Trusted Resource Domain Controller

void TRDC_Init(TRDC_Type *base)

Initializes the TRDC module.

This function enables the TRDC clock.

Parameters

- base – TRDC peripheral base address.

`void TRDC_Deinit(TRDC_Type *base)`

De-initializes the TRDC module.

This function disables the TRDC clock.

Parameters

- `base` – TRDC peripheral base address.

`static inline uint8_t TRDC_GetCurrentMasterDomainId(TRDC_Type *base)`

Gets the domain ID of the current bus master.

Parameters

- `base` – TRDC peripheral base address.

Returns

Domain ID of current bus master.

`void TRDC_GetHardwareConfig(TRDC_Type *base, trdc_hardware_config_t *config)`

Gets the TRDC hardware configuration.

This function gets the TRDC hardware configurations, including number of bus masters, number of domains, number of MRCs and number of PACs.

Parameters

- `base` – TRDC peripheral base address.
- `config` – Pointer to the structure to get the configuration.

`static inline void TRDC_SetDacGlobalValid(TRDC_Type *base)`

Sets the TRDC DAC(Domain Assignment Controllers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- `base` – TRDC peripheral base address.

`static inline void TRDC_LockMasterDomainAssignment(TRDC_Type *base, uint8_t master)`

Locks the bus master domain assignment register.

This function locks the master domain assignment. After it is locked, the register can't be changed until next reset.

Parameters

- `base` – TRDC peripheral base address.
- `master` – Which master to configure.

`static inline void TRDC_SetMasterDomainAssignmentValid(TRDC_Type *base, uint8_t master, bool valid)`

Sets the master domain assignment as valid or invalid.

This function sets the master domain assignment as valid or invalid.

Parameters

- `base` – TRDC peripheral base address.
- `master` – Which master to configure.
- `valid` – True to set valid, false to set invalid.

```
void TRDC_GetDefaultProcessorDomainAssignment(trdc_processor_domain_assignment_t
                                              *domainAssignment)
```

Gets the default master domain assignment for the processor bus master.

This function gets the default master domain assignment for the processor bus master. It should only be used for the processor bus masters, such as CORE0. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->domainIdSelect = kTRDC_DidMda;
assignment->lock           = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

```
void TRDC_GetDefaultNonProcessorDomainAssignment(trdc_non_processor_domain_assignment_t
                                                  *domainAssignment)
```

Gets the default master domain assignment for non-processor bus master.

This function gets the default master domain assignment for non-processor bus master. It should only be used for the non-processor bus masters, such as DMA. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->privilegeAttr  = kTRDC_ForceUser;
assignment->secureAttr     = kTRDC_ForceSecure;
assignment->bypassDomainId = 0U;
assignment->lock           = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

```
void TRDC_SetProcessorDomainAssignment(TRDC_Type *base, const
                                       trdc_processor_domain_assignment_t
                                       *domainAssignment)
```

Sets the processor bus master domain assignment.

This function sets the processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for core 0.

```
trdc_processor_domain_assignment_t processorAssignment;

TRDC_GetDefaultProcessorDomainAssignment(&processorAssignment);

processorAssignment.domainId = 0;
processorAssignment.xxx      = xxx;
TRDC_SetMasterDomainAssignment(TRDC, &processorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- domainAssignment – Pointer to the assignment structure.

```
static inline void TRDC_EnableProcessorDomainAssignment(TRDC_Type *base, bool enable)
```

Enables the processor bus master domain assignment.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

```
void TRDC_SetNonProcessorDomainAssignment(TRDC_Type *base, uint8_t master, const
                                         trdc_non_processor_domain_assignment_t
                                         *domainAssignment)
```

Sets the non-processor bus master domain assignment.

This function sets the non-processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for DMA0.

```
trdc_non_processor_domain_assignment_t nonProcessorAssignment;

TRDC_GetDefaultNonProcessorDomainAssignment(&nonProcessorAssignment);
nonProcessorAssignment.domainId = 1;
nonProcessorAssignment.xxx      = xxx;

TRDC_SetMasterDomainAssignment(TRDC, kTrdcMasterDma0, 0U, &nonProcessorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- master – Which master to configure, refer to `trdc_master_t` in processor header file.
- domainAssignment – Pointer to the assignment structure.

```
void TRDC_GetDefaultIDAUConfig(trdc_idau_config_t *idauConfiguration)
```

Gets the default IDAU(Implementation-Defined Attribution Unit) configuration.

```
config->lockSecureVTOR   = false;
config->lockNonsecureVTOR = false;
config->lockSecureMPU     = false;
config->lockNonsecureMPU  = false;
config->lockSAU           = false;
```

Parameters

- idauConfiguration – Pointer to the configuration structure.

```
void TRDC_SetIDAU(TRDC_Type *base, const trdc_idau_config_t *idauConfiguration)
```

Sets the IDAU(Implementation-Defined Attribution Unit) control configuration.

Example: Lock the secure and non-secure MPU registers.

```
trdc_idau_config_t idauConfiguration;

TRDC_GetDefaultIDAUConfig(&idauConfiguration);

idauConfiguration.lockSecureMPU = true;
idauConfiguration.lockNonsecureMPU = true;
TRDC_SetIDAU(TRDC, &idauConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- idauConfiguration – Pointer to the configuration structure.

static inline void TRDC_EnableFlashLogicalWindow(TRDC_Type *base, bool enable)
Enables/disables the FLW(flash logical window) function.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

static inline void TRDC_LockFlashLogicalWindow(TRDC_Type *base)
Locks FLW registers. Once locked the registers can not be updated until next reset.

Parameters

- base – TRDC peripheral base address.

static inline uint32_t TRDC_GetFlashLogicalWindowPbase(TRDC_Type *base)
Gets the FLW physical base address.

Parameters

- base – TRDC peripheral base address.

Returns

Physical address of the FLW function.

static inline void TRDC_GetSetFlashLogicalWindowSize(TRDC_Type *base, uint16_t size)
Sets the FLW size.

Parameters

- base – TRDC peripheral base address.
- size – Size of the FLW in unit of 32k bytes.

void TRDC_GetDefaultFlashLogicalWindowConfig(*trdc_flw_config_t* *flwConfiguration)
Gets the default FLW(Flash Logical Window) configuration.

```
config->blockCount = false;
config->arrayBaseAddr = false;
config->lock = false;
config->enable = false;
```

Parameters

- flwConfiguration – Pointer to the configuration structure.

void TRDC_SetFlashLogicalWindow(TRDC_Type *base, const *trdc_flw_config_t* *flwConfiguration)
Sets the FLW function's configuration.

```
trdc_flw_config_t flwConfiguration;

TRDC_GetDefaultIDAUConfig(&flwConfiguration);

flwConfiguration.blockCount = 32U;
flwConfiguration.arrayBaseAddr = 0xFFFFFFFF;
TRDC_SetIDAU(TRDC, &flwConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- flwConfiguration – Pointer to the configuration structure.

status_t TRDC_GetAndClearFirstDomainError(TRDC_Type *base, *trdc_domain_error_t* *error)

Gets and clears the first domain error of the current domain.

This function gets the first access violation information for the current domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.

Returns

If the access violation is captured, this function returns the *kStatus_Success*. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the *kStatus_NoData*.

status_t TRDC_GetAndClearFirstSpecificDomainError(TRDC_Type *base, *trdc_domain_error_t* *error, *uint8_t* domainId)

Gets and clears the first domain error of the specific domain.

This function gets the first access violation information for the specific domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.
- domainId – The error of which domain to get and clear.

Returns

If the access violation is captured, this function returns the *kStatus_Success*. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the *kStatus_NoData*.

static inline void TRDC_SetMrcGlobalValid(TRDC_Type *base)

Sets the TRDC MRC(Memory Region Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

static inline *uint8_t* TRDC_GetMrcRegionNumber(TRDC_Type *base, *uint8_t* mrcIdx)

Gets the TRDC MRC(Memory Region Checkers) region number valid.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.

Returns

the region number of the given MRC instance

void TRDC_MrcSetMemoryAccessConfig(TRDC_Type *base, const *trdc_memory_access_control_config_t* *config, *uint8_t* mrcIdx, *uint8_t* regIdx)

Sets the memory access configuration for one of the access control register of one MRC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;

config.securePrivX = true;
config.securePrivW = true;
config.securePrivR = true;
config.lock = true;
TRDC_SetMrcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mrcIdx – MRC index.
- regIdx – Register number.

void TRDC_MrcEnableDomainNseUpdate(TRDC_Type *base, uint8_t mrcIdx, uint16_t domainMask, bool enable)

Enables the update of the selected domains.

After the domains' update are enabled, their regions' NSE bits can be set or clear.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains to be enabled.
- enable – True to enable, false to disable.

void TRDC_MrcRegionNseSet(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)

Sets the NSE bits of the selected regions for domains.

This function sets the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to set.

void TRDC_MrcRegionNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)

Clears the NSE bits of the selected regions for domains.

This function clears the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to clear.

void TRDC_MrcDomainNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t domainMask)

Clears the NSE bits for all the regions of the selected domains.

This function clears the NSE bits for all regions of selected domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains whose NSE bits to clear.

```
void TRDC__MrcSetRegionDescriptorConfig(TRDC_Type *base, const
                                         trdc_mrc_region_descriptor_config_t *config)
```

Sets the configuration for one of the region descriptor per domain per MRC instance.

This function sets the configuration for one of the region descriptor, including the start and end address of the region, memory access control policy and valid.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to region descriptor configuration structure.

```
static inline void TRDC__SetMbcGlobalValid(TRDC_Type *base)
```

Sets the TRDC MBC(Memory Block Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

```
void TRDC__GetMbcHardwareConfig(TRDC_Type *base, trdc_slave_memory_hardware_config_t
                                *config, uint8_t mbcIdx, uint8_t slvIdx)
```

Gets the hardware configuration of the one of two slave memories within each MBC(memory block checker).

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the structure to get the configuration.
- mbcIdx – MBC number.
- slvIdx – Slave number.

```
void TRDC__MbcSetNseUpdateConfig(TRDC_Type *base, const trdc_mbc_nse_update_config_t
                                  *config, uint8_t mbcIdx)
```

Sets the NSR update configuration for one of the MBC instance.

After set the NSE configuration, the configured memory area can be updateby NSE set/clear.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to NSE update configuration structure.
- mbcIdx – MBC index.

```
void TRDC__MbcWordNseSet(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)
```

Sets the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to set.


```
void TRDC_MbcWordNseClear(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)
```

Clears the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to clear.

```
void TRDC_MbcNseClearAll(TRDC_Type *base, uint8_t mbcIdx, uint16_t domainMask, uint8_t slaveMask)
```

Clears all configuration words' NSE bits of the selected domain and memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- domainMask – Mask of the domains whose NSE bits to clear, 0b110 means clear domain 1&2.
- slaveMask – Mask of the slaves whose NSE bits to clear, 0x11 means clear all slave 0&1's NSE bits.

```
void TRDC_MbcSetMemoryAccessConfig(TRDC_Type *base, const
                                   trdc_memory_access_control_config_t *config, uint8_t
                                   mbcIdx, uint8_t rgdIdx)
```

Sets the memory access configuration for one of the region descriptor of one MBC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;

config.securePrivX = true;
config.securePrivW = true;
config.securePrivR = true;
config.lock = true;
TRDC_SetMbcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mbcIdx – MBC index.
- rgdIdx – Region descriptor number.

```
void TRDC_MbcSetMemoryBlockConfig(TRDC_Type *base, const
                                  trdc_mbc_memory_block_config_t *config)
```

Sets the configuration for one of the memory block per domain per MBC instance.

This function sets the configuration for one of the memory block, including the memory access control policy and nse enable.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to memory block configuration structure.

enum _trdc_did_sel

TRDC domain ID select method, the register bit TRDC_MDA_W0_0_DFMT0[DIDS], used for domain hit evaluation.

Values:

enumerator kTRDC_DidMda

Use MDAn[2:0] as DID.

enumerator kTRDC_DidInput

Use the input DID (DID_in) as DID.

enumerator kTRDC_DidMdaAndInput

Use MDAn[2] concatenated with DID_in[1:0] as DID.

enumerator kTRDC_DidReserved

Reserved.

enum _trdc_secure_attr

TRDC secure attribute, the register bit TRDC_MDA_W0_0_DFMT0[SA], used for bus master domain assignment.

Values:

enumerator kTRDC_ForceSecure

Force the bus attribute for this master to secure.

enumerator kTRDC_ForceNonSecure

Force the bus attribute for this master to non-secure.

enumerator kTRDC_MasterSecure

Use the bus master's secure/nonsecure attribute directly.

enumerator kTRDC_MasterSecure1

Use the bus master's secure/nonsecure attribute directly.

enum _trdc_privilege_attr

TRDC privileged attribute, the register bit TRDC_MDA_W0_x_DFMT1[PA], used for non-processor bus master domain assignment.

Values:

enumerator kTRDC_ForceUser

Force the bus attribute for this master to user.

enumerator kTRDC_ForcePrivilege

Force the bus attribute for this master to privileged.

enumerator kTRDC_MasterPrivilege

Use the bus master's attribute directly.

enumerator kTRDC_MasterPrivilege1

Use the bus master's attribute directly.

enum _trdc_controller

TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call TRDC_GetHardwareConfig to get the actual count.

Values:

enumerator kTRDC_MemBlockController0

Memory block checker 0.

enumerator kTRDC_MemBlockController1
Memory block checker 1.

enumerator kTRDC_MemBlockController2
Memory block checker 2.

enumerator kTRDC_MemBlockController3
Memory block checker 3.

enumerator kTRDC_MemRegionChecker0
Memory region checker 0.

enumerator kTRDC_MemRegionChecker1
Memory region checker 1.

enumerator kTRDC_MemRegionChecker2
Memory region checker 2.

enumerator kTRDC_MemRegionChecker3
Memory region checker 3.

enumerator kTRDC_MemRegionChecker4
Memory region checker 4.

enumerator kTRDC_MemRegionChecker5
Memory region checker 5.

enumerator kTRDC_MemRegionChecker6
Memory region checker 6.

enum _trdc_error_state

TRDC domain error state	definition	TRDC_MBCn_DERR_W1[EST]	or
TRDC_MRCn_DERR_W1[EST].			

Values:

enumerator kTRDC_ErrorStateNone
No access violation detected.

enumerator kTRDC_ErrorStateNone1
No access violation detected.

enumerator kTRDC_ErrorStateSingle
Single access violation detected.

enumerator kTRDC_ErrorStateMulti
Multiple access violation detected.

enum _trdc_error_attr

TRDC domain error attribute	definition	TRDC_MBCn_DERR_W1[EATR]	or
TRDC_MRCn_DERR_W1[EATR].			

Values:

enumerator kTRDC_ErrorSecureUserInst
Secure user mode, instruction fetch access.

enumerator kTRDC_ErrorSecureUserData
Secure user mode, data access.

enumerator kTRDC_ErrorSecurePrivilegeInst
Secure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorSecurePrivilegeData
Secure privileged mode, data access.

enumerator kTRDC_ErrorNonSecureUserInst
NonSecure user mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecureUserData
NonSecure user mode, data access.

enumerator kTRDC_ErrorNonSecurePrivilegeInst
NonSecure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecurePrivilegeData
NonSecure privileged mode, data access.

enum _trdc_error_type
TRDC domain error access type definition TRDC_DERR_W1_n[ERW].

Values:

enumerator kTRDC_ErrorTypeRead
Error occurs on read reference.

enumerator kTRDC_ErrorTypeWrite
Error occurs on write reference.

enum _trdc_region_descriptor
The region descriptor enumeration, used to form a mask to set/clear the NSE bits for one or several regions.

Values:

enumerator kTRDC_RegionDescriptor0
Region descriptor 0.

enumerator kTRDC_RegionDescriptor1
Region descriptor 1.

enumerator kTRDC_RegionDescriptor2
Region descriptor 2.

enumerator kTRDC_RegionDescriptor3
Region descriptor 3.

enumerator kTRDC_RegionDescriptor4
Region descriptor 4.

enumerator kTRDC_RegionDescriptor5
Region descriptor 5.

enumerator kTRDC_RegionDescriptor6
Region descriptor 6.

enumerator kTRDC_RegionDescriptor7
Region descriptor 7.

enumerator kTRDC_RegionDescriptor8
Region descriptor 8.

enumerator kTRDC_RegionDescriptor9
Region descriptor 9.

enumerator kTRDC_RegionDescriptor10
Region descriptor 10.

enumerator kTRDC_RegionDescriptor11
Region descriptor 11.

enumerator kTRDC_RegionDescriptor12
Region descriptor 12.

enumerator kTRDC_RegionDescriptor13
Region descriptor 13.

enumerator kTRDC_RegionDescriptor14
Region descriptor 14.

enumerator kTRDC_RegionDescriptor15
Region descriptor 15.

enum _trdc_MRC_domain

The MRC domain enumeration, used to form a mask to enable/disable the update or clear all NSE bits of one or several domains.

Values:

enumerator kTRDC_MrcDomain0
Domain 0.

enumerator kTRDC_MrcDomain1
Domain 1.

enumerator kTRDC_MrcDomain2
Domain 2.

enumerator kTRDC_MrcDomain3
Domain 3.

enumerator kTRDC_MrcDomain4
Domain 4.

enumerator kTRDC_MrcDomain5
Domain 5.

enumerator kTRDC_MrcDomain6
Domain 6.

enumerator kTRDC_MrcDomain7
Domain 7.

enumerator kTRDC_MrcDomain8
Domain 8.

enumerator kTRDC_MrcDomain9
Domain 9.

enumerator kTRDC_MrcDomain10
Domain 10.

enumerator kTRDC_MrcDomain11
Domain 11.

enumerator kTRDC_MrcDomain12
Domain 12.

enumerator kTRDC_MrcDomain13
Domain 13.

enumerator kTRDC_MrcDomain14

Domain 14.

enumerator kTRDC_MrcDomain15

Domain 15.

enum _trdc_MBC_domain

The MBC domain enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several domains.

Values:

enumerator kTRDC_MbcDomain0

Domain 0.

enumerator kTRDC_MbcDomain1

Domain 1.

enumerator kTRDC_MbcDomain2

Domain 2.

enumerator kTRDC_MbcDomain3

Domain 3.

enumerator kTRDC_MbcDomain4

Domain 4.

enumerator kTRDC_MbcDomain5

Domain 5.

enumerator kTRDC_MbcDomain6

Domain 6.

enumerator kTRDC_MbcDomain7

Domain 7.

enum _trdc_MBC_memory

The MBC slave memory enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several memory block.

Values:

enumerator kTRDC_MbcSlaveMemory0

Memory 0.

enumerator kTRDC_MbcSlaveMemory1

Memory 1.

enumerator kTRDC_MbcSlaveMemory2

Memory 2.

enumerator kTRDC_MbcSlaveMemory3

Memory 3.

enum _trdc_MBC_bit

The MBC bit enumeration, used to form a mask to set/clear configured words' NSE.

Values:

enumerator kTRDC_MbcBit0

Bit 0.

enumerator kTRDC_MbcBit1

Bit 1.

enumerator kTRDC_MbcBit2
Bit 2.

enumerator kTRDC_MbcBit3
Bit 3.

enumerator kTRDC_MbcBit4
Bit 4.

enumerator kTRDC_MbcBit5
Bit 5.

enumerator kTRDC_MbcBit6
Bit 6.

enumerator kTRDC_MbcBit7
Bit 7.

enumerator kTRDC_MbcBit8
Bit 8.

enumerator kTRDC_MbcBit9
Bit 9.

enumerator kTRDC_MbcBit10
Bit 10.

enumerator kTRDC_MbcBit11
Bit 11.

enumerator kTRDC_MbcBit12
Bit 12.

enumerator kTRDC_MbcBit13
Bit 13.

enumerator kTRDC_MbcBit14
Bit 14.

enumerator kTRDC_MbcBit15
Bit 15.

enumerator kTRDC_MbcBit16
Bit 16.

enumerator kTRDC_MbcBit17
Bit 17.

enumerator kTRDC_MbcBit18
Bit 18.

enumerator kTRDC_MbcBit19
Bit 19.

enumerator kTRDC_MbcBit20
Bit 20.

enumerator kTRDC_MbcBit21
Bit 21.

enumerator kTRDC_MbcBit22
Bit 22.

enumerator `kTRDC_MbcBit23`

Bit 23.

enumerator `kTRDC_MbcBit24`

Bit 24.

enumerator `kTRDC_MbcBit25`

Bit 25.

enumerator `kTRDC_MbcBit26`

Bit 26.

enumerator `kTRDC_MbcBit27`

Bit 27.

enumerator `kTRDC_MbcBit28`

Bit 28.

enumerator `kTRDC_MbcBit29`

Bit 29.

enumerator `kTRDC_MbcBit30`

Bit 30.

enumerator `kTRDC_MbcBit31`

Bit 31.

typedef struct *_trdc_hardware_config* trdc_hardware_config_t

TRDC hardware configuration.

typedef struct *_trdc_slave_memory_hardware_config* trdc_slave_memory_hardware_config_t

Hardware configuration of the two slave memories within each MBC(memory block checker).

typedef enum *_trdc_did_sel* trdc_did_sel_t

TRDC domain ID select method, the register bit `TRDC_MDA_W0_0_DFMT0[DIDS]`, used for domain hit evaluation.

typedef enum *_trdc_secure_attr* trdc_secure_attr_t

TRDC secure attribute, the register bit `TRDC_MDA_W0_0_DFMT0[SA]`, used for bus master domain assignment.

typedef struct *_trdc_processor_domain_assignment* trdc_processor_domain_assignment_t

Domain assignment for the processor bus master.

typedef enum *_trdc_privilege_attr* trdc_privilege_attr_t

TRDC privileged attribute, the register bit `TRDC_MDA_W0_x_DFMT1[PA]`, used for non-processor bus master domain assignment.

typedef struct *_trdc_non_processor_domain_assignment*

trdc_non_processor_domain_assignment_t

Domain assignment for the non-processor bus master.

typedef struct *_trdc_idau_config* trdc_idau_config_t

IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

typedef struct *_trdc_flw_config* trdc_flw_config_t

FLW(Flash Logical Window) configuration.

typedef enum *_trdc_controller* trdc_controller_t

TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call `TRDC_GetHardwareConfig` to get the actual count.


```
typedef enum _trdc_error_state trdc_error_state_t
    TRDC domain error state definition TRDC_MBCn_DERR_W1[EST] or
    TRDC_MRCn_DERR_W1[EST].
```

```
typedef enum _trdc_error_attr trdc_error_attr_t
    TRDC domain error attribute definition TRDC_MBCn_DERR_W1[EATR] or
    TRDC_MRCn_DERR_W1[EATR].
```

```
typedef enum _trdc_error_type trdc_error_type_t
    TRDC domain error access type definition TRDC_DERR_W1_n[ERW].
```

```
typedef struct _trdc_domain_error trdc_domain_error_t
    TRDC domain error definition.
```

```
typedef struct _trdc_memory_access_control_config trdc_memory_access_control_config_t
    Memory access control configuration for MBC/MRC.
```

```
typedef struct _trdc_mrc_region_descriptor_config trdc_mrc_region_descriptor_config_t
    The configuration of each region descriptor per domain per MRC instance.
```

```
typedef struct _trdc_mbc_nse_update_config trdc_mbc_nse_update_config_t
    The configuration of MBC NSE update.
```

```
typedef struct _trdc_mbc_memory_block_config trdc_mbc_memory_block_config_t
    The configuration of each memory block per domain per MBC instance.
```

```
FSL_TRDC_DRIVER_VERSION
```

```
struct _trdc_hardware_config
    #include <fsl_trdc.h> TRDC hardware configuration.
```

Public Members

```
uint8_t masterNumber
    Number of bus masters.
```

```
uint8_t domainNumber
    Number of domains.
```

```
uint8_t mbcNumber
    Number of MBCs.
```

```
uint8_t mrcNumber
    Number of MRCs.
```

```
struct _trdc_slave_memory_hardware_config
    #include <fsl_trdc.h> Hardware configuration of the two slave memories within each
    MBC(memory block checker).
```

Public Members

```
uint32_t blockNum
    Number of blocks.
```

```
uint32_t blockSize
    Block size.
```

```
struct _trdc_processor_domain_assignment
    #include <fsl_trdc.h> Domain assignment for the processor bus master.
```

Public Members

uint32_t domainId

Domain ID.

uint32_t domainIdSelect

Domain ID select method, see trdc_did_sel_t.

uint32_t __pad0__

Reserved.

uint32_t secureAttr

Secure attribute, see trdc_secure_attr_t.

uint32_t __pad1__

Reserved.

uint32_t lock

Lock the register.

uint32_t __pad2__

Reserved.

struct __trdc_non_processor_domain_assignment

#include <fsl_trdc.h> Domain assignment for the non-processor bus master.

Public Members

uint32_t domainId

Domain ID.

uint32_t privilegeAttr

Privileged attribute, see trdc_privilege_attr_t.

uint32_t secureAttr

Secure attribute, see trdc_secure_attr_t.

uint32_t bypassDomainId

Bypass domain ID.

uint32_t __pad0__

Reserved.

uint32_t lock

Lock the register.

uint32_t __pad1__

Reserved.

struct __trdc_idau_config

#include <fsl_trdc.h> IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

Public Members

uint32_t __pad0__

Reserved.

uint32_t lockSecureVTOR

Disable writes to secure VTOR(Vector Table Offset Register).

uint32_t lockNonsecureVTOR

Disable writes to non-secure VTOR, Application interrupt and Reset Control Registers.

uint32_t lockSecureMPU

Disable writes to secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor in Secure state.

uint32_t lockNonsecureMPU

Disable writes to non-secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor.

uint32_t lockSAU

Disable writes to SAU(Security Attribution Unit) registers.

uint32_t __pad1__

Reserved.

struct __trdc_flw_config

#include <fsl_trdc.h> FLW(Flash Logical Window) configuration.

Public Members

uint16_t blockCount

Block count of the Flash Logic Window in 32KByte blocks.

uint32_t arrayBaseAddr

Flash array base address of the Flash Logical Window.

bool lock

Disable writes to FLW registers.

bool enable

Enable FLW function.

struct __trdc_domain_error

#include <fsl_trdc.h> TRDC domain error definition.

Public Members

trdc_controller_t controller

Which controller captured access violation.

uint32_t address

Access address that generated access violation.

trdc_error_state_t errorState

Error state.

trdc_error_attr_t errorAttr

Error attribute.

trdc_error_type_t errorType

Error type.

uint8_t errorPort

Error port.

uint8_t domainId

Domain ID.

uint8_t slaveMemoryIdx

The slave memory index. Only apply when violation in MBC.

struct __trdc_memory_access_control_config

#include <fsl_trdc.h> Memory access control configuration for MBC/MRC.

Public Members

uint32_t nonsecureUsrX

Allow nonsecure user execute access.

uint32_t nonsecureUsrW

Allow nonsecure user write access.

uint32_t nonsecureUsrR

Allow nonsecure user read access.

uint32_t __pad0__

Reserved.

uint32_t nonsecurePrivX

Allow nonsecure privilege execute access.

uint32_t nonsecurePrivW

Allow nonsecure privilege write access.

uint32_t nonsecurePrivR

Allow nonsecure privilege read access.

uint32_t __pad1__

Reserved.

uint32_t secureUsrX

Allow secure user execute access.

uint32_t secureUsrW

Allow secure user write access.

uint32_t secureUsrR

Allow secure user read access.

uint32_t __pad2__

Reserved.

uint32_t securePrivX

Allownonsecure privilege execute access.

uint32_t securePrivW

Allownonsecure privilege write access.

uint32_t securePrivR

Allownonsecure privilege read access.

uint32_t __pad3__

Reserved.

uint32_t lock

Lock the configuration until next reset, only apply to access control register 0.

struct __trdc_mrc_region_descriptor_config

#include <fsl_trdc.h> The configuration of each region descriptor per domain per MRC instance.

Public Members

`uint8_t` `memoryAccessControlSelect`
Select one of the 8 access control policies for this region, for access control policies see `trdc_memory_access_control_config_t`.

`uint32_t` `startAddr`
Physical start address.

`bool` `valid`
Lock the register.

`bool` `nseEnable`
Enable non-secure accesses and disable secure accesses.

`uint32_t` `endAddr`
Physical start address.

`uint8_t` `mrcIdx`
The index of the MRC for this configuration to take effect.

`uint8_t` `domainIdx`
The index of the domain for this configuration to take effect.

`uint8_t` `regionIdx`
The index of the region for this configuration to take effect.

`struct` `_trdc_mbc_nse_update_config`
#include <fsl_trdc.h> The configuration of MBC NSE update.

Public Members

`uint32_t` `__pad0__`
Reserved.

`uint32_t` `wordIdx`
MBC configuration word index to be updated.

`uint32_t` `__pad1__`
Reserved.

`uint32_t` `memorySelect`
Bit mask of the selected memory to be updated. `_trdc_MBC_memory`.

`uint32_t` `__pad2__`
Reserved.

`uint32_t` `domainSelect`
Bit mask of the selected domain to be updated. `_trdc_MBC_domain`.

`uint32_t` `__pad3__`
Reserved.

`uint32_t` `autoIncrement`
Whether to increment the word index after current word is updated using this configuration.

`struct` `_trdc_mbc_memory_block_config`
#include <fsl_trdc.h> The configuration of each memory block per domain per MBC instance.

Public Members

uint32_t memoryAccessControlSelect

Select one of the 8 access control policies for this memory block, for access control policies see `trdc_memory_access_control_config_t`.

uint32_t nseEnable

Enable non-secure accesses and disable secure accesses.

uint32_t mbcIdx

The index of the MBC for this configuration to take effect.

uint32_t domainIdx

The index of the domain for this configuration to take effect.

uint32_t slaveMemoryIdx

The index of the slave memory for this configuration to take effect.

uint32_t memoryBlockIdx

The index of the memory block for this configuration to take effect.

2.71 TRGMUX: Trigger Mux Driver

static inline void TRGMUX_LockRegister(TRGMUX_Type *base, uint32_t index)

Sets the flag of the register which is used to mark writeable.

The function sets the flag of the register which is used to mark writeable. Example:

```
TRGMUX_LockRegister(TRGMUX0, kTRGMUX_Trgmux0Dmamux0);
```

Parameters

- base – TRGMUX peripheral base address.
- index – The index of the TRGMUX register, see the enum `trgmux_device_t` defined in `<SOC>.h`.

status_t TRGMUX_SetTriggerSource(TRGMUX_Type *base, uint32_t index,
trgmux_trigger_input_t input, uint32_t trigger_src)

Configures the trigger source of the appointed peripheral.

The function configures the trigger source of the appointed peripheral. Example:

```
TRGMUX_SetTriggerSource(TRGMUX0, kTRGMUX_Trgmux0Dmamux0, kTRGMUX_TriggerInput0,  
↪ kTRGMUX_SourcePortPin);
```

Parameters

- base – TRGMUX peripheral base address.
- index – The index of the TRGMUX register, see the enum `trgmux_device_t` defined in `<SOC>.h`.
- input – The MUX select for peripheral trigger input
- trigger_src – The trigger inputs for various peripherals. See the enum `trgmux_source_t` defined in `<SOC>.h`.

Return values

- kStatus_Success – Configured successfully.

- `kStatus_TRGMUX_Locked` – Configuration failed because the register is locked.

`FSL_TRGMUX_DRIVER_VERSION`

TRGMUX driver version.

TRGMUX configure status.

Values:

enumerator `kStatus_TRGMUX_Locked`

Configure failed for register is locked

enum `_trgmux_trigger_input`

Defines the MUX select for peripheral trigger input.

Values:

enumerator `kTRGMUX_TriggerInput0`

The MUX select for peripheral trigger input 0

enumerator `kTRGMUX_TriggerInput1`

The MUX select for peripheral trigger input 1

enumerator `kTRGMUX_TriggerInput2`

The MUX select for peripheral trigger input 2

enumerator `kTRGMUX_TriggerInput3`

The MUX select for peripheral trigger input 3

typedef enum `_trgmux_trigger_input` `trgmux_trigger_input_t`

Defines the MUX select for peripheral trigger input.

2.72 TSTMR: Timestamp Timer Driver

void `TSTMR_Init(TSTMR_Type *base)`

Init TSTMR.

This function initializes the TSTMR module.

Parameters

- `base` – TSTMR peripheral base address.

void `TSTMR_Deinit(TSTMR_Type *base)`

Deinit TSTMR.

This function deinitializes the TSTMR module.

Parameters

- `base` – TSTMR peripheral base address.

`FSL_TSTMR_DRIVER_VERSION`

Version 2.1.0

static inline uint64_t `TSTMR_ReadTimeStamp(TSTMR_Type *base)`

Reads the time stamp.

This function reads the low and high registers and returns the 56-bit free running counter value. This can be read by software at any time to determine the software ticks. TSTMR registers can be read with 32-bit accesses only. The TSTMR LOW read should occur first, followed by the TSTMR HIGH read.

Parameters

- base – TSTMR peripheral base address.

Returns

The 56-bit time stamp value.

void TSTMR_DelayUs(TSTMR_Type *base, uint64_t delayInUs)

Delays for a specified number of microseconds.

This function repeatedly reads the timestamp register and waits for the user-specified delay value.

Parameters

- base – TSTMR peripheral base address.
- delayInUs – Delay value in microseconds.

2.73 VBAT: Smart Power Switch

The enumeration of VBAT module status.

Values:

enumerator kStatus_VBAT_Fro16kNotEnabled

Internal 16kHz free running oscillator not enabled.

enumerator kStatus_VBAT_BandgapNotEnabled

Bandgap not enabled.

enum __vbat_status_flag

The enumeration of VBAT status flags.

Values:

enumerator kVBAT_StatusFlagPORDetect

VBAT domain has been reset

enumerator kVBAT_StatusFlagWakeupPin

A falling edge is detected on the wakeup pin.

enumerator kVBAT_StatusFlagBandgapTimer0

Bandgap Timer0 period reached.

enumerator kVBAT_StatusFlagBandgapTimer1

Bandgap Timer1 period reached.

enumerator kVBAT_StatusFlagLdoReady

LDO is enabled and ready.

enum __vbat_interrupt_enable

The enumeration of VBAT interrupt enable.

Values:

enumerator kVBAT_InterruptEnablePORDetect

Enable POR detect interrupt.

enumerator kVBAT_InterruptEnableWakeupPin

Enable the interrupt when a falling edge is detected on the wakeup pin.

enumerator kVBAT__InterruptEnableBandgapTimer0

Enable the interrupt if Bandgap Timer0 period reached.

enumerator kVBAT__InterruptEnableBandgapTimer1

Enable the interrupt if Bandgap Timer1 period reached.

enumerator kVBAT__InterruptEnableLdoReady

Enable LDO ready interrupt.

enumerator kVBAT__AllInterruptsEnable

Enable all interrupts.

enum __vbat__wakeup__enable

The enumeration of VBAT wakeup enable.

Values:

enumerator kVBAT__WakeupEnablePORDetect

Enable POR detect wakeup.

enumerator kVBAT__WakeupEnableWakeupPin

Enable wakeup feature when a falling edge is detected on the wakeup pin.

enumerator kVBAT__WakeupEnableBandgapTimer0

Enable wakeup feature when bandgap timer0 period reached.

enumerator kVBAT__WakeupEnableBandgapTimer1

Enable wakeup feature when bandgap timer1 period reached.

enumerator kVBAT__WakeupEnableLdoReady

Enable wakeup when LDO ready.

enumerator kVBAT__AllWakeupsEnable

Enable all wakeup.

enum __vbat__bandgap__timer__id

The enumeration of bandgap timer id, VBAT support two bandgap timers.

Values:

enumerator kVBAT__BandgapTimer0

Bandgap Timer0.

enumerator kVBAT__BandgapTimer1

Bandgap Timer1.

enum __vbat__bandgap__refresh__period

The enumeration of bandgap refresh period.

Values:

enumerator kVBAT__BandgapRefresh7P8125ms

Bandgap refresh every 7.8125ms.

enumerator kVBAT__BandgapRefresh15P625ms

Bandgap refresh every 15.625ms.

enumerator kVBAT__BandgapRefresh31P25ms

Bandgap refresh every 31.25ms.

enumerator kVBAT__BandgapRefresh62P5ms

Bandgap refresh every 62.5ms.

enum `_vbat_bandgap_timer_timeout_period`

The enumeration of bandgap timer timeout period.

Values:

enumerator `kVBAT_BangapTimerTimeout1s`

Bandgap timer timerout every 1s.

enumerator `kVBAT_BangapTimerTimeout500ms`

Bandgap timer timerout every 500ms.

enumerator `kVBAT_BangapTimerTimeout250ms`

Bandgap timer timerout every 250ms.

enumerator `kVBAT_BangapTimerTimeout125ms`

Bandgap timer timerout every 125ms.

enumerator `kVBAT_BangapTimerTimeout62P5ms`

Bandgap timer timerout every 62.5ms.

enumerator `kVBAT_BangapTimerTimeout31P25ms`

Bandgap timer timerout every 31.25ms.

typedef enum `_vbat_bandgap_refresh_period` `vbat_bandgap_refresh_period_t`

The enumeration of bandgap refresh period.

typedef enum `_vbat_bandgap_timer_timeout_period` `vbat_bandgap_timer_timeout_period_t`

The enumeration of bandgap timer timeout period.

typedef struct `_vbat_fro16k_config` `vbat_fro16k_config_t`

The structure of internal 16kHz free running oscillator attributes.

`VBAT_LDO_READY_TIMEOUT`

Max loops to wait for LDO ready.

When configuring the LDO, driver will wait for LDO ready. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

void `VBAT_ConfigFRO16k(VBAT_Type *base, const vbat_fro16k_config_t *config)`

Configure internal 16kHz free running oscillator, including enabel FRO16k, gate FRO16k output.

Parameters

- `base` – VBAT peripheral base address.
- `config` – Pointer to `vbat_fro16k_config_t` structure.

static inline void `VBAT_EnableFRO16k(VBAT_Type *base, bool enable)`

Enable/disable internal 16kHz free running oscillator.

Parameters

- `base` – VBAT peripheral base address.
- `enable` – Used to enable/disable 16kHz FRO.
 - **true** Enable internal 16kHz free running oscillator.
 - **false** Disable internal 16kHz free running oscillator.

static inline bool `VBAT_CheckFRO16kEnabled(VBAT_Type *base)`

Check if internal 16kHz free running oscillator is enabled.

Parameters

- `base` – VBAT peripheral base address.

Return values

- `true` – The internal 16kHz Free running oscillator is enabled.
- `false` – The internal 16kHz Free running oscillator is enabled.

`static inline void VBAT__UngateFRO16k(VBAT_Type *base, bool unGateFRO16k)`

Ungate/gate FRO 16kHz output clock to other modules.

Parameters

- `base` – VBAT peripheral base address.
- `unGateFRO16k` – Used to gate/ungate FRO 16kHz output.
 - **true** FRO 16kHz output clock to other modules is enabled.
 - **false** FRO 16kHz output clock to other modules is disabled.

`static inline void VBAT__LockFRO16kSettings(VBAT_Type *base)`

Lock settings of internal 16kHz free running oscillator, please note that if locked 16kHz FRO's settings can not be updated until the next POR.

Note: Please note that the operation to ungate/gate FRO 16kHz output clock can not be locked by this function.

Parameters

- `base` – VBAT peripheral base address.

`status_t VBAT__EnableBandgap(VBAT_Type *base, bool enable)`

Enable/disable Bandgap.

Note: The FRO16K must be enabled before enabling the bandgap.

Note: This setting can be locked by `VBAT__LockLdoRamSettings()` function.

Parameters

- `base` – VBAT peripheral base address.
- `enable` – Used to enable/disable bandgap.
 - **true** Enable the bandgap.
 - **false** Disable the bandgap.

Return values

- `kStatus__Success` – Success to enable/disable the bandgap.
- `kStatus__VBAT__Fro16kNotEnabled` – Fail to enable the bandgap due to FRO16k is not enabled previously.

`static inline bool VBAT__CheckBandgapEnabled(VBAT_Type *base)`

Check if bandgap is enabled.

Parameters

- `base` – VBAT peripheral base address.

Return values

- `true` – The bandgap is enabled.
- `false` – The bandgap is disabled.

`static inline void VBAT__EnableBandgapRefreshMode(VBAT_Type *base, bool enableRefreshMode)`

Enable/disable bandgap low power refresh mode.

Note: This setting can be locked by `VBAT_LockLdoRamSettings()` function.

Parameters

- `base` – VBAT peripheral base address.
- `enableRefreshMode` – Used to enable/disable bandgap low power refresh mode.
 - **true** Enable bandgap low power refresh mode.
 - **false** Disable bandgap low power refresh mode.

`status_t VBAT__EnableBackupSRAMRegulator(VBAT_Type *base, bool enable)`
Enable/disable Backup RAM Regulator(RAM_LDO).

Note: This setting can be locked by `VBAT_LockLdoRamSettings()` function.

Parameters

- `base` – VBAT peripheral base address.
- `enable` – Used to enable/disable RAM_LDO.
 - **true** Enable backup SRAM regulator.
 - **false** Disable backup SRAM regulator.

Return values

- `kStatusSuccess` – Success to enable/disable backup SRAM regulator.
- `kStatus_VBAT__Fro16kNotEnabled` – Fail to enable backup SRAM regulator due to FRO16k is not enabled previously.
- `kStatus_VBAT__BandgapNotEnabled` – Fail to enable backup SRAM regulator due to the bandgap is not enabled previously.
- `kStatus_Timeout` – Timeout occurs while waiting completion.

`static inline void VBAT__LockLdoRamSettings(VBAT_Type *base)`

Lock settings of RAM_LDO, please note that if locked then RAM_LDO's settings can not be updated until the next POR.

Parameters

- `base` – VBAT peripheral base address.

`status_t VBAT__SwitchSRAMPowerByVBAT(VBAT_Type *base)`

Switch the SRAM to be powered by VBAT in software mode.

Note: This function can be used to switch the SRAM to the VBAT retention supply at any time, but please note that the SRAM must not be accessed during this time and software must manually invoke `VBAT_SwitchSRAMPowerBySocSupply()` before accessing the SRAM again.

Parameters

- base – VBAT peripheral base address.

Return values

- kStatusSuccess – Success to Switch SRAM powered by VBAT.
- kStatus_VBAT_Fro16kNotEnabled – Fail to switch SRAM powered by VBAT due to FRO16K not enabled previously.

```
static inline void VBAT__SwitchSRAMPowerBySocSupply(VBAT_Type *base)
```

Switch the RAM to be powered by Soc Supply in software mode.

Parameters

- base – VBAT peripheral base address.

```
static inline void VBAT__EnableSRAMArrayRetained(VBAT_Type *base, bool enable)
```

Enable/disable SRAM array remains powered from Soc power, when LDO_RAM is disabled.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable SRAM array power retained.
 - **true** SRAM array is retained when powered from VDD_CORE.
 - **false** SRAM array is not retained when powered from VDD_CORE.

```
static inline void VBAT__EnableSRAMIsolation(VBAT_Type *base, bool enable)
```

Enable/disable SRAM isolation.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable SRAM violation.
 - **true** SRAM will be isolated.
 - **false** SRAM state follows the SoC power modes.

```
status_t VBAT__EnableBandgapTimer(VBAT_Type *base, bool enable, uint8_t timerIdMask)
```

Enable/disable Bandgap timer.

Note: The bandgap timer is available when the bandgap is enabled and are clocked by the FRO16k.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable bandgap timer.
- timerIdMask – The mask of bandgap timer Id, should be the OR'ed value of vbat_bandgap_timer_id_t.

Return values

- kStatus__Success – Success to enable/disable selected bandgap timer.
- kStatus_VBAT_Fro16kNotEnabled – Fail to enable/disable selected bandgap timer due to FRO16k not enabled previously.
- kStatus_VBAT_BandgapNotEnabled – Fail to enable/disable selected bandgap timer due to bandgap not enabled previously.

```
void VBAT_SetBandgapTimerTimeoutValue(VBAT_Type *base,  
                                     vbat_bandgap_timer_timeout_period_t  
                                     timeoutPeriod, uint8_t timerIdMask)
```

Set bandgap timer timeout value.

Parameters

- base – VBAT peripheral base address.
- timeoutPeriod – Bandgap timer timeout value, please refer to `vbat_bandgap_timer_timeout_period_t`.
- timerIdMask – The mask of bandgap timer Id, should be the OR'ed value of `vbat_bandgap_timer_id_t`.

```
static inline uint32_t VBAT_GetStatusFlags(VBAT_Type *base)
```

Get VBAT status flags.

Parameters

- base – VBAT peripheral base address.

Returns

The asserted status flags, should be the OR'ed value of `vbat_status_flag_t`.

```
static inline void VBAT_ClearStatusFlags(VBAT_Type *base, uint32_t mask)
```

Clear VBAT status flags.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of status flags to be cleared, should be the OR'ed value of `vbat_status_flag_t` except `kVBAT_StatusFlagLdoReady`.

```
static inline void VBAT_EnableInterrupts(VBAT_Type *base, uint32_t mask)
```

Enable interrupts for the VBAT module, such as POR detect interrupt, Wakeup Pin interrupt and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of interrupts to be enabled, should be the OR'ed value of `vbat_interrupt_enable_t`.

```
static inline void VBAT_DisableInterrupts(VBAT_Type *base, uint32_t mask)
```

Disable interrupts for the VBAT module, such as POR detect interrupt, wakeup pin interrupt and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of interrupts to be disabled, should be the OR'ed value of `vbat_interrupt_enable_t`.

```
static inline void VBAT_EnableWakeup(VBAT_Type *base, uint32_t mask)
```

Enable wakeup for the VBAT module, such as POR detect wakeup, wakeup pin wakeup and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of enumerators in `vbat_wakeup_enable_t`.

```
static inline void VBAT_DisableWakeup(VBAT_Type *base, uint32_t mask)
```

Disable wakeup for VBAT module, such as POR detect wakeup, wakeup pin wakeup and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of enumerators in vbat_wakeup_enable_t.

```
static inline void VBAT_LockInterruptWakeupSettings(VBAT_Type *base)
```

Lock VBAT interrupt and wakeup settings, please note that if locked the interrupt and wakeup settings can not be updated until the next POR.

Parameters

- base – VBAT peripheral base address.

```
FSL_VBAT_DRIVER_VERSION
```

VBAT driver version 2.1.1.

```
struct _vbat_fro16k_config
```

#include <fsl_vbat.h> The structure of internal 16kHz free running oscillator attributes.

Public Members

```
bool enableFRO16k
```

Enable/disable internal 16kHz free running oscillator.

```
bool enableFRO16kOutput
```

Enable/disable FRO 16k output clock to other modules.

2.74 VREF: Voltage Reference Driver

```
void VREF_Init(VREF_Type *base, const vref_config_t *config)
```

Enables the clock gate and configures the VREF module according to the configuration structure.

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up vref_config_t parameters and how to call the VREF_Init function by passing in these parameters.

```
vref_config_t vrefConfig;
VREF_GetDefaultConfig(&vrefConfig);
vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
VREF_Init(VREF, &vrefConfig);
```

Parameters

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

```
void VREF_Deinit(VREF_Type *base)
```

Stops and disables the clock for the VREF module.

This function should be called to shut down the module. This is an example.

```
vref_config_t vrefUserConfig;  
VREF_GetDefaultConfig(&vrefUserConfig);  
VREF_Init(VREF, &vrefUserConfig);  
...  
VREF_Deinit(VREF);
```

Parameters

- base – VREF peripheral address.

void VREF_GetDefaultConfig(*vref_config_t* *config)

Initializes the VREF configuration structure.

This function initializes the VREF configuration structure to default values. This is an example.

```
config->bufferMode = kVREF_ModeHighPowerBuffer;  
config->enableInternalVoltageRegulator = true;  
config->enableChopOscillator           = true;  
config->enableHCBandgap                 = true;  
config->enableCurvatureCompensation    = true;  
config->enableLowPowerBuff              = true;
```

Parameters

- config – Pointer to the initialization structure.

void VREF_SetVrefTrimVal(VREF_Type *base, uint8_t trimValue)

Sets a TRIM value for the accurate 1.0V bandgap output.

This function sets a TRIM value for the reference voltage. It will trim the accurate 1.0V bandgap by 0.5mV each step.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

void VREF_SetTrim21Val(VREF_Type *base, uint8_t trim21Value)

Sets a TRIM value for the accurate buffered VREF output.

This function sets a TRIM value for the reference voltage. If buffer mode be set to other values (Buf21 enabled), it will trim the VREF_OUT by 0.1V each step from 1.0V to 2.1V.

Note: When Buf21 is enabled, the value of UTRIM[TRIM2V1] should be ranged from 0b0000 to 0b1011 in order to trim the output voltage from 1.0V to 2.1V, other values will make the VREF_OUT to default value, 1.0V.

Parameters

- base – VREF peripheral address.
- trim21Value – Value of the trim register to set the output reference voltage (maximum 0xF (4-bit)).

uint8_t VREF_GetVrefTrimVal(VREF_Type *base)

Reads the trim value.

This function gets the TRIM value from the UTRIM register. It reads UTRIM[VREFTRIM] (13:8)

Parameters

- base – VREF peripheral address.

Returns

6-bit value of trim setting.

uint8_t VREF_GetTrim21Val(VREF_Type *base)

Reads the VREF 2.1V trim value.

This function gets the TRIM value from the UTRIM register. It reads UTRIM[TRIM2V1] (3:0),

Parameters

- base – VREF peripheral address.

Returns

4-bit value of trim setting.

FSL_VREF_DRIVER_VERSION

Version 2.4.0.

enum __vref_buffer_mode

VREF buffer modes.

Values:

enumerator kVREF_ModeBandgapOnly

Bandgap enabled/standby.

enumerator kVREF_ModeLowPowerBuffer

Low-power buffer mode enabled

enumerator kVREF_ModeHighPowerBuffer

High-power buffer mode enabled

typedef enum __vref_buffer_mode vref_buffer_mode_t

VREF buffer modes.

typedef struct __vref_config vref_config_t

The description structure for the VREF module.

struct __vref_config

#include <fsl_vref.h> The description structure for the VREF module.

Public Members

vref_buffer_mode_t bufferMode

Buffer mode selection

bool enableInternalVoltageRegulator

Provide additional supply noise rejection.

bool enableChopOscillator

Enable Chop oscillator.

bool enableHCBandgap

Enable High-Accurate bandgap.

bool enableCurvatureCompensation

Enable second order curvature compensation.

bool enableLowPowerBuff

Provides bias current for other peripherals.

2.75 WDOG32: 32-bit Watchdog Timer

`void WDOG32_GetDefaultConfig(wdog32_config_t *config)`

Initializes the WDOG32 configuration structure.

This function initializes the WDOG32 configuration structure to default values. The default values are:

```
wdog32Config->enableWdog32 = true;
wdog32Config->clockSource = kWDOG32_ClockSource1;
wdog32Config->prescaler = kWDOG32_ClockPrescalerDivide1;
wdog32Config->workMode.enableWait = true;
wdog32Config->workMode.enableStop = false;
wdog32Config->workMode.enableDebug = false;
wdog32Config->testMode = kWDOG32_TestModeDisabled;
wdog32Config->enableUpdate = true;
wdog32Config->enableInterrupt = false;
wdog32Config->enableWindowMode = false;
wdog32Config->windowValue = 0U;
wdog32Config->timeoutValue = 0xFFFFU;
```

See also:

`wdog32_config_t`

Parameters

- `config` – Pointer to the WDOG32 configuration structure.

`status_t WDOG32_Init(WDOG_Type *base, const wdog32_config_t *config)`

Initializes the WDOG32 module.

This function initializes the WDOG32. To reconfigure the WDOG32 without forcing a reset first, `enableUpdate` must be set to true in the configuration.

Example:

```
wdog32_config_t config;
WDOG32_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG32_Init(wdog_base,&config);
```

Note: If there is errata ERR010536 (FSL_FEATURE_WDOG_HAS_ERRATA_010536 defined as 1), then after calling this function, user need delay at least 4 LPO clock cycles before accessing other WDOG32 registers.

Parameters

- `base` – WDOG32 peripheral base address.
- `config` – The configuration of the WDOG32.

Return values

- `kStatus_Success` – The initialization was successful
- `kStatus_Timeout` – The initialization timed out

status_t WDOG32_Deinit(WDOG_Type *base)

De-initializes the WDOG32 module.

This function shuts down the WDOG32. Ensure that the WDOG_CS.UPDATE is 1, which means that the register update is enabled.

Parameters

- base – WDOG32 peripheral base address.

Return values

- kStatus_Success – The de-initialization was successful
- kStatus_Timeout – The de-initialization timed out

status_t WDOG32_Unlock(WDOG_Type *base)

Unlocks the WDOG32 register written.

This function unlocks the WDOG32 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG32 peripheral base address

Return values

- kStatus_Success – The unlock sequence was successful
- kStatus_Timeout – The unlock sequence timed out

void WDOG32_Enable(WDOG_Type *base)

Enables the WDOG32 module.

This function writes a value into the WDOG_CS register to enable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.

void WDOG32_Disable(WDOG_Type *base)

Disables the WDOG32 module.

This function writes a value into the WDOG_CS register to disable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address

void WDOG32_EnableInterrupts(WDOG_Type *base, uint32_t mask)

Enables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to enable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- mask – The interrupts to enable. The parameter can be a combination of the following source if defined:
 - kWDOG32_InterruptEnable

`void WDOG32_DisableInterrupts(WDOG_Type *base, uint32_t mask)`
Disables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to disable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- mask – The interrupts to disabled. The parameter can be a combination of the following source if defined:
 - kWDOG32_InterruptEnable

`static inline uint32_t WDOG32_GetStatusFlags(WDOG_Type *base)`
Gets the WDOG32 all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;  
status = WDOG32_GetStatusFlags(wdog_base) & kWDOG32_RunningFlag;
```

See also:

`_wdog32_status_flags_t`

- true: related status flag has been set.
- false: related status flag is not set.

Parameters

- base – WDOG32 peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

`void WDOG32_ClearStatusFlags(WDOG_Type *base, uint32_t mask)`
Clears the WDOG32 flag.

This function clears the WDOG32 status flag.

Example to clear an interrupt flag:

```
WDOG32_ClearStatusFlags(wdog_base, kWDOG32_InterruptFlag);
```

Parameters

- base – WDOG32 peripheral base address.
- mask – The status flags to clear. The parameter can be any combination of the following values:
 - kWDOG32_InterruptFlag

`void WDOG32_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)`

Sets the WDOG32 timeout value.

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. To ensure the reconfiguration fits the timing of WCT, unlock function will be called inline.

Parameters

- `base` – WDOG32 peripheral base address
- `timeoutCount` – WDOG32 timeout value, count of WDOG32 clock ticks.

`void WDOG32_SetWindowValue(WDOG_Type *base, uint16_t windowValue)`

Sets the WDOG32 window value.

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Please check the `enableUpdate` is set to true for calling `WDOG32_Init` to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG32 peripheral base address.
- `windowValue` – WDOG32 window value.

`static inline void WDOG32_Refresh(WDOG_Type *base)`

Refreshes the WDOG32 timer.

This function feeds the WDOG32. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- `base` – WDOG32 peripheral base address

`static inline uint16_t WDOG32_GetCounterValue(WDOG_Type *base)`

Gets the WDOG32 counter value.

This function gets the WDOG32 counter value.

Parameters

- `base` – WDOG32 peripheral base address.

Returns

Current WDOG32 counter value.

`WDOG_FIRST_WORD_OF_UNLOCK`

First word of unlock sequence

`WDOG_SECOND_WORD_OF_UNLOCK`

Second word of unlock sequence

`WDOG_FIRST_WORD_OF_REFRESH`

First word of refresh sequence

`WDOG_SECOND_WORD_OF_REFRESH`

Second word of refresh sequence

`FSL_WDOG32_DRIVER_VERSION`

WDOG32 driver version.

enum `_wdog32_clock_source`

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

Values:

enumerator `kWDOG32_ClockSource0`

Clock source 0

enumerator `kWDOG32_ClockSource1`

Clock source 1

enumerator `kWDOG32_ClockSource2`

Clock source 2

enumerator `kWDOG32_ClockSource3`

Clock source 3

enum `_wdog32_clock_prescaler`

Describes the selection of the clock prescaler.

Values:

enumerator `kWDOG32_ClockPrescalerDivide1`

Divided by 1

enumerator `kWDOG32_ClockPrescalerDivide256`

Divided by 256

enum `_wdog32_test_mode`

Describes WDOG32 test mode.

Values:

enumerator `kWDOG32_TestModeDisabled`

Test Mode disabled

enumerator `kWDOG32_UserModeEnabled`

User Mode enabled

enumerator `kWDOG32_LowByteTest`

Test Mode enabled, only low byte is used

enumerator `kWDOG32_HighByteTest`

Test Mode enabled, only high byte is used

enum `_wdog32_interrupt_enable_t`

WDOG32 interrupt configuration structure.

This structure contains the settings for all of the WDOG32 interrupt configurations.

Values:

enumerator `kWDOG32_InterruptEnable`

Interrupt is generated before forcing a reset

enum `_wdog32_status_flags_t`

WDOG32 status flags.

This structure contains the WDOG32 status flags for use in the WDOG32 functions.

Values:

enumerator `kWDOG32_RunningFlag`

Running flag, set when WDOG32 is enabled

enumerator `kWDOG32_InterruptFlag`

Interrupt flag, set when interrupt occurs

typedef enum `_wdog32_clock_source` `wdog32_clock_source_t`

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

typedef enum `_wdog32_clock_prescaler` `wdog32_clock_prescaler_t`

Describes the selection of the clock prescaler.

typedef struct `_wdog32_work_mode` `wdog32_work_mode_t`

Defines WDOG32 work mode.

typedef enum `_wdog32_test_mode` `wdog32_test_mode_t`

Describes WDOG32 test mode.

typedef struct `_wdog32_config` `wdog32_config_t`

Describes WDOG32 configuration structure.

struct `_wdog32_work_mode`

`#include <fsl_wdog32.h>` Defines WDOG32 work mode.

Public Members

bool `enableWait`

Enables or disables WDOG32 in wait mode

bool `enableStop`

Enables or disables WDOG32 in stop mode

bool `enableDebug`

Enables or disables WDOG32 in debug mode

struct `_wdog32_config`

`#include <fsl_wdog32.h>` Describes WDOG32 configuration structure.

Public Members

bool `enableWdog32`

Enables or disables WDOG32

`wdog32_clock_source_t` `clockSource`

Clock source select

wdog32_clock_prescaler_t prescaler

Clock prescaler value

wdog32_work_mode_t workMode

Configures WDOG32 work mode in debug stop and wait mode

wdog32_test_mode_t testMode

Configures WDOG32 test mode

bool enableUpdate

Update write-once register enable

bool enableInterrupt

Enables or disables WDOG32 interrupt

bool enableWindowMode

Enables or disables WDOG32 window mode

uint16_t windowValue

Window value

uint16_t timeoutValue

Timeout value

2.76 WUU: Wakeup Unit driver

void WUU_SetExternalWakeUpPinsConfig(WUU_Type *base, uint8_t pinIndex, const
wuu_external_wakeup_pin_config_t *config)

Enables and Configs External WakeUp Pins.

This function enables/disables the external pin as wakeup input. What's more this function configs pins options, including edge detection wakeup event and operate mode.

Parameters

- base – MUU peripheral base address.
- pinIndex – The index of the external input pin. See Reference Manual for the details.
- config – Pointer to *wuu_external_wakeup_pin_config_t* structure.

void WUU_ClearExternalWakeupPinsConfig(WUU_Type *base, uint8_t pinIndex)

Disable and clear external wakeup pin settings.

Parameters

- base – MUU peripheral base address.
- pinIndex – The index of the external input pin.

static inline uint32_t WUU_GetExternalWakeUpPinsFlag(WUU_Type *base)

Gets External Wakeup pin flags.

This function return the external wakeup pin flags.

Parameters

- base – WUU peripheral base address.

Returns

Wakeup flags for all external wakeup pins.


```
static inline void WUU_ClearExternalWakeUpPinsFlag(WUU_Type *base, uint32_t mask)
```

Clears External WakeUp Pin flags.

This function clears external wakeup pins flags based on the mask.

Parameters

- base – WUU peripheral base address.
- mask – The mask of Wakeup pin index to be cleared.

```
void WUU_SetInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex,  
                                         wuu_internal_wakeup_module_event_t event)
```

Config Internal modules' event as the wake up sources.

This function configs the internal modules event as the wake up sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – Select interrupt or DMA/Trigger of the internal module as the wake up source.

```
void WUU_ClearInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex,  
                                           wuu_internal_wakeup_module_event_t event)
```

Disable an on-chip internal modules' event as the wakeup sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – The event(interrupt or DMA/trigger) of the internal module to disable.

```
void WUU_SetPinFilterConfig(WUU_Type *base, uint8_t filterIndex, const  
                            wuu_pin_filter_config_t *config)
```

Configs and Enables Pin filters.

This function configs Pin filter, including pin select, filter operate mode filter wakeup event and filter edge detection.

Parameters

- base – WUU peripheral base address.
- filterIndex – The index of the pin filter.
- config – Pointer to *wuu_pin_filter_config_t* structure.

```
bool WUU_GetPinFilterFlag(WUU_Type *base, uint8_t filterIndex)
```

Gets the pin filter configuration.

This function gets the pin filter flag.

Parameters

- base – WUU peripheral base address.
- filterIndex – A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

void WUU_ClearPinFilterFlag(WUU_Type *base, uint8_t filterIndex)

Clears the pin filter configuration.

This function clears the pin filter flag.

Parameters

- base – WUU peripheral base address.
- filterIndex – A pin filter index to clear the flag, starting from 1.

bool WUU_GetExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)

brief Gets the external wakeup source flag.

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

param base WUU peripheral base address. param pinIndex A pin index, which starts from 0. return True if the specific pin is a wakeup source.

void WUU_ClearExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)

brief Clears the external wakeup source flag.

This function clears the external wakeup source flag for a specific pin.

param base WUU peripheral base address. param pinIndex A pin index, which starts from 0.

FSL_WUU_DRIVER_VERSION

Defines WUU driver version 2.4.1.

enum _wuu_external_pin_edge_detection

External WakeUp pin edge detection enumeration.

Values:

enumerator kWUU_ExternalPinDisable

External input Pin disabled as wake up input.

enumerator kWUU_ExternalPinRisingEdge

External input Pin enabled with the rising edge detection.

enumerator kWUU_ExternalPinFallingEdge

External input Pin enabled with the falling edge detection.

enumerator kWUU_ExternalPinAnyEdge

External input Pin enabled with any change detection.

enum _wuu_external_wakeup_pin_event

External input wake up pin event enumeration.

Values:

enumerator kWUU_ExternalPinInterrupt

External input Pin configured as interrupt.

enumerator kWUU_ExternalPinDMARequest

External input Pin configured as DMA request.

enumerator kWUU_ExternalPinTriggerEvent

External input Pin configured as Trigger event.

enum _wuu_external_wakeup_pin_mode

External input wake up pin mode enumeration.

Values:

enumerator kWUU_ExternalPinActiveDSPD

External input Pin is active only during Deep Sleep/Power Down Mode. NOTE: This enumerations has been deprecated, please switch to kWUU_ExternalPinActiveLowLeakage.

enumerator kWUU_ExternalPinActiveLowLeakageMode

External input Pin is active only during low-leakage power modes.

enumerator kWUU_ExternalPinActiveAlways

External input Pin is active during all power modes.

enum _wuu_internal_wakeup_module_event

Internal module wake up event enumeration.

Values:

enumerator kWUU_InternalModuleInterrupt

Internal modules' interrupt as a wakeup source.

enumerator kWUU_InternalModuleDMATrigger

Internal modules' DMA/Trigger as a wakeup source.

enum _wuu_filter_edge

Pin filter edge enumeration.

Values:

enumerator kWUU_FilterDisabled

Filter disabled.

enumerator kWUU_FilterPosedgeEnable

Filter posedge detect enabled.

enumerator kWUU_FilterNegedgeEnable

Filter negedge detect enabled.

enumerator kWUU_FilterAnyEdge

Filter any edge detect enabled.

enum _wuu_filter_event

Pin Filter event enumeration.

Values:

enumerator kWUU_FilterInterrupt

Filter output configured as interrupt.

enumerator kWUU_FilterDMARequest

Filter output configured as DMA request.

enumerator kWUU_FilterTriggerEvent

Filter output configured as Trigger event.

enum _wuu_filter_mode

Pin filter mode enumeration.

Values:

enumerator kWUU_FilterActiveDSPD

External input pin filter is active only during Deep Sleep/Power Down Mode. NOTE: This enumerations has been deprecated, please switch to kWUU_FilterActiveLowLeakage.

enumerator kWUU_FilterActiveLowLeakageMode

External input pin filter is active only during low-leakage power modes.

enumerator kWUU_FilterActiveAlways

External input Pin filter is active during all power modes.

typedef enum *_wuU_external_pin_edge_detection* wuu_external_pin_edge_detection_t

External WakeUp pin edge detection enumeration.

typedef enum *_wuU_external_wakeup_pin_event* wuu_external_wakeup_pin_event_t

External input wake up pin event enumeration.

typedef enum *_wuU_external_wakeup_pin_mode* wuu_external_wakeup_pin_mode_t

External input wake up pin mode enumeration.

typedef enum *_wuU_internal_wakeup_module_event* wuu_internal_wakeup_module_event_t

Internal module wake up event enumeration.

typedef enum *_wuU_filter_edge* wuu_filter_edge_t

Pin filter edge enumeration.

typedef enum *_wuU_filter_event* wuu_filter_event_t

Pin Filter event enumeration.

typedef enum *_wuU_filter_mode* wuu_filter_mode_t

Pin filter mode enumeration.

typedef struct *_wuU_external_wakeup_pin_config* wuu_external_wakeup_pin_config_t

External WakeUp pin configuration.

typedef struct *_wuU_pin_filter_config* wuu_pin_filter_config_t

Pin Filter configuration.

struct *_wuU_external_wakeup_pin_config*

#include <fsl_wuu.h> External WakeUp pin configuration.

Public Members

wuu_external_pin_edge_detection_t edge

External Input pin edge detection.

wuu_external_wakeup_pin_event_t event

External Input wakeup Pin event

wuu_external_wakeup_pin_mode_t mode

External Input wakeup Pin operate mode.

struct *_wuU_pin_filter_config*

#include <fsl_wuu.h> Pin Filter configuration.

Public Members

uint32_t pinIndex

The index of wakeup pin to be muxxed into filter.

wuu_filter_edge_t edge

The edge of the pin digital filter.

wuu_filter_event_t event

The event of the filter output.

wuu_filter_mode_t mode

The mode of the filter operate.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called “packet-driven BDM” interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT](<https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/>) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to [FreeMASTER community](#) or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a “middleware” component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER “middleware” driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample 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 *fmaster*: 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.

3.2 MultiCore

3.2.1 Multicore SDK

Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

Multicore SDK (MCSDK) Release Notes

Overview These are the release notes for the NXP Multicore Software Development Kit (MCSDK) version 25.12.00.

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

What is new

- eRPC [CHANGELOG](#)
- RPPMsg-Lite [CHANGELOG](#)
- MCMgr [CHANGELOG](#)
- Supported evaluation boards (multicore examples):
 - LPCXpresso55S69
 - FRDM-K32L3A6
 - MIMXRT1170-EVKB
 - MIMXRT1160-EVK
 - MIMXRT1180-EVK
 - MCX-N5XX-EVK
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK
 - KW47-EVK
 - KW47-LOC
 - FRDM-MCXW72
 - MCX-W72-EVK
 - FRDM-IMXRT1186
- Supported evaluation boards (multiprocessor examples):
 - LPCXpresso55S36
 - FRDM-K22F
 - FRDM-K32L2B
 - MIMXRT685-EVK
 - MIMXRT1170-EVKB
 - MIMXRT1180
 - FRDM-MCXN236
 - FRDM-MCXC242
 - FRDM-MCXC444
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK
 - FRDM-IMXRT1186

Development tools The Multicore SDK (MCSDK) was compiled and tested with development tools referred in: [Development tools](#)

Release contents This table describes the release contents. Not all MCUXpresso SDK packages contain the whole set of these components.

Deliverable	Location
Multicore SDK location	<MCUXpressoSDK_install_dir>/middleware/multicore/
Documentation	<MCSDK_dir>/mcuxsdk-doc/
Embedded Remote Procedure Call component	<MCSDK_dir>/erpc/
Multicore Manager component	<MCSDK_dir>/mcmgr/
RPMsg-Lite	<MCSDK_dir>/rpmsg_lite/
Multicore demo applications	<MCUXpressoSDK_install_dir>/examples/multicore_examples/
Multiprocessor demo applications	<MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/

Multicore SDK release overview Together, the Multicore SDK (MCSDK) and the MCUXpresso SDK (SDK) form a framework for the development of software for NXP multicore devices. The MCSDK release consists of the following elementary software components for multicore:

- Embedded Remote Procedure Call (eRPC)
- Multicore Manager (MCMGR) - included just in SDK for multicore devices
- Remote Processor Messaging - Lite (RPMsg-Lite) - included just in SDK for multicore devices

The MCSDK is also accompanied with documentation and several multicore and multiprocessor demo applications.

Demo applications The multicore demo applications demonstrate the usage of the MCSDK software components on supported multicore development boards.

The following multicore demo applications are located together with other MCUXpresso SDK examples in the <MCUXpressoSDK_install_dir>/examples/multicore_examples subdirectories.

- erpc_matrix_multiply_mu
- erpc_matrix_multiply_mu_rtos
- erpc_matrix_multiply_rpmsg
- erpc_matrix_multiply_rpmsg_rtos
- erpc_two_way_rpc_rpmsg_rtos
- freertos_message_buffers
- hello_world
- multicore_manager
- rpmsg_lite_pingpong
- rpmsg_lite_pingpong_rtos
- rpmsg_lite_pingpong_dsp
- rpmsg_lite_pingpong_tzm

The eRPC multicore component can be leveraged for inter-processor communication and remote procedure calls between SoCs / development boards.

The following multiprocessor demo applications are located together with other MCUXpresso SDK examples in

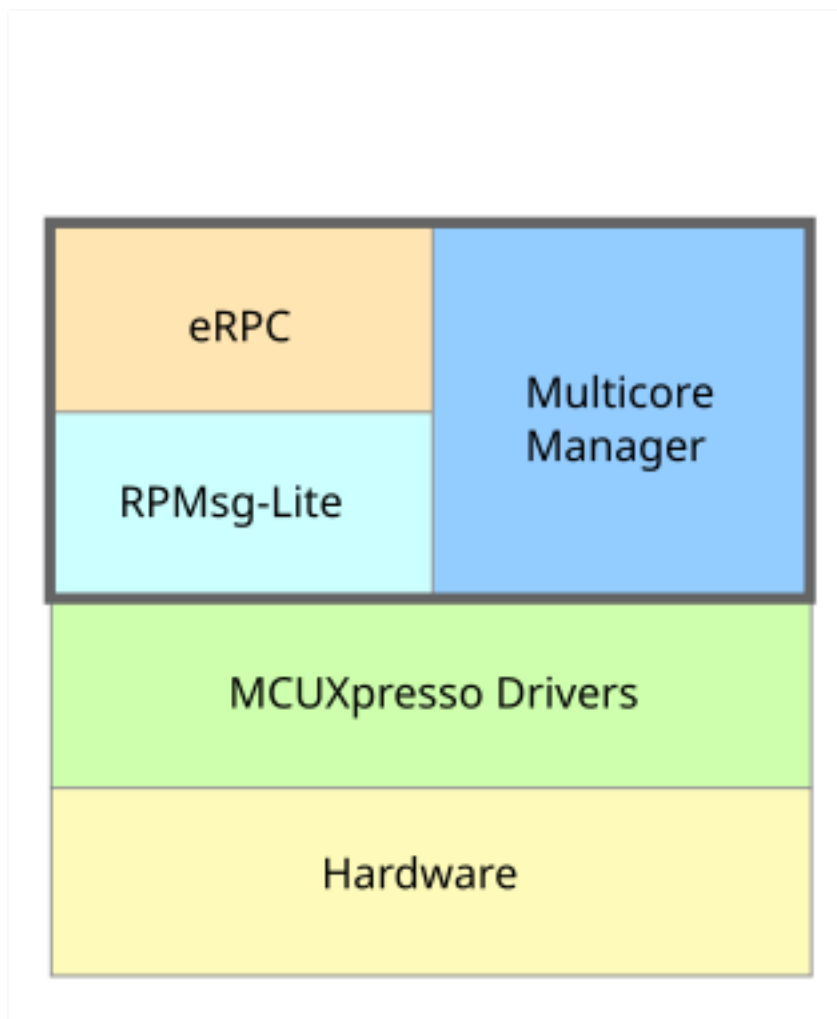
the <MCUXpressoSDK_install_dir>/examples/multiprocessor_examples subdirectories.

- erpc_client_matrix_multiply_spi
- erpc_server_matrix_multiply_spi
- erpc_client_matrix_multiply_uart
- erpc_server_matrix_multiply_uart
- erpc_server_dac_adc
- erpc_remote_control

Getting Started with Multicore SDK (MCSDK)

Overview Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

The following figure highlights the layers and main software components of the MCSDK.

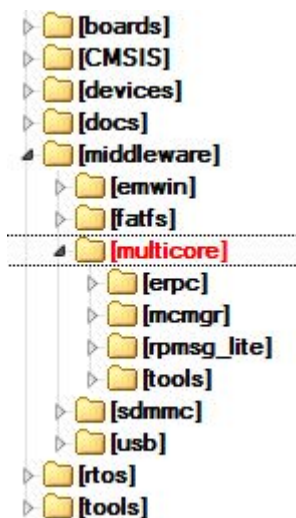


All the MCSDK-related files are located in `<MCUXpressoSDK_install_dir>/middleware/multicore` folder.

For supported toolchain versions, see the *Multicore SDK v25.12.00 Release Notes* (document MCS-DKRN). For the latest version of this and other MCSDK documents, visit www.nxp.com.

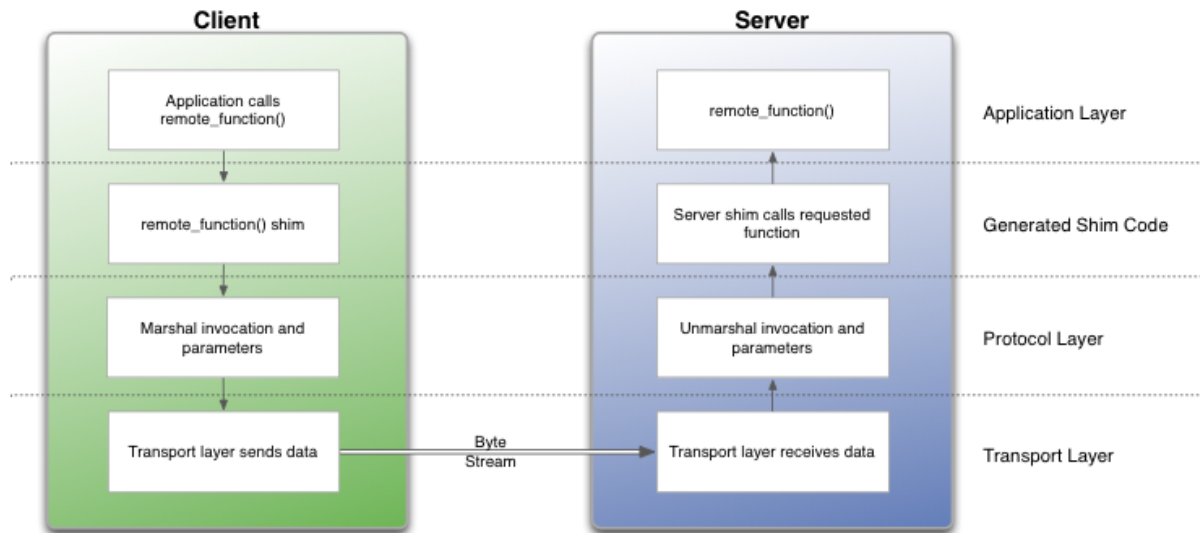
Multicore SDK (MCSDK) components The MCSDK consists of the following software components:

- **Embedded Remote Procedure Call (eRPC):** This component is a combination of a library and code generator tool that implements a transparent function call interface to remote services (running on a different core).
- **Multicore Manager (MCMGR):** This library maintains information about all cores and starts up secondary/auxiliary cores.
- **Remote Processor Messaging - Lite (RPMsg-Lite):** Inter-Processor Communication library.



Embedded Remote Procedure Call (eRPC) The Embedded Remote Procedure Call (eRPC) is the RPC system created by NXP. The RPC is a mechanism used to invoke a software routine on a remote system via a simple local function call.

When a remote function is called by the client, the function's parameters and an identifier for the called routine are marshaled (or serialized) into a stream of bytes. This byte stream is transported to the server through a communications channel (IPC, TPC/IP, UART, and so on). The server unmarshals the parameters, determines which function was invoked, and calls it. If the function returns a value, it is marshaled and sent back to the client.



RPC implementations typically use a combination of a tool (erpcgen) and IDL (interface definition language) file to generate source code to handle the details of marshaling a function's parameters and building the data stream.

Main eRPC features:

- Scalable from BareMetal to Linux OS - configurable memory and threading policies.
- Focus on embedded systems - intrinsic support for C, modular, and lightweight implementation.
- Abstracted transport interface - RPMsg is the primary transport for multicore, UART, or SPI-based solutions can be used for multichip.

The eRPC library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc` folder. For detailed information about the eRPC, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems.

The main MCMGR features:

- Maintains information about all cores in system.
- Secondary/auxiliary cores startup and shutdown.
- Remote core monitoring and event handling.

The MCMGR library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr` folder. For detailed information about the MCMGR library, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/doc` folder.

Remote Processor Messaging Lite (RPMsg-Lite) RPMsg-Lite is a lightweight implementation of the RPMsg protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. Compared to the legacy OpenAMP implementation, RPMsg-Lite offers a code size reduction, API simplification, and improved modularity.

The main RPMsg protocol features:

- Shared memory interprocessor communication.
- Virtio-based messaging bus.
- Application-defined messages sent between endpoints.

- Portable to different environments/platforms.
- Available in upstream Linux OS.

The RPMsg-Lite library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg-lite` folder. For detailed information about the RPMsg-Lite, see the RPMsg-Lite User's Guide located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/doc` folder.

MCSDK demo applications Multicore and multiprocessor example applications are stored together with other MCUXpresso SDK examples, in the dedicated multicore subfolder.

Location		Folder
Multicore projects	example	<code><MCUXpressoSDK_install_dir>/examples/multicore_examples/<application_name>/</code>
Multiprocessor projects	example	<code><MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/<application_name>/</code>

See the *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) and *Getting Started with MCUXpresso SDK for XXX Derivatives* documents for more information about the MCUXpresso SDK example folder structure and the location of individual files that form the example application projects. These documents also contain information about building, running, and debugging multicore demo applications in individual supported IDEs. Each example application also contains a readme file that describes the operation of the example and required setup steps.

Inter-Processor Communication (IPC) levels The MCSDK provides several mechanisms for Inter-Processor Communication (IPC). Particular ways and levels of IPC are described in this chapter.

IPC using low-level drivers

The NXP multicore SoCs are equipped with peripheral modules dedicated for data exchange between individual cores. They deal with the Mailbox peripheral for LPC parts and the Messaging Unit (MU) peripheral for Kinetis and i.MX parts. The common attribute of both modules is the ability to provide a means of IPC, allowing multiple CPUs to share resources and communicate with each other in a simple manner.

The most lightweight method of IPC uses the MCUXpresso SDK low-level drivers for these peripherals. Using the Mailbox/MU driver API functions, it is possible to pass a value from core to core via the dedicated registers (could be a scalar or a pointer to shared memory) and also to trigger inter-core interrupts for notifications.

For details about individual driver API functions, see the MCUXpresso SDK API Reference Manual of the specific multicore device. The MCUXpresso SDK is accompanied with the RPMsg-Lite documentation that shows how to use this API in multicore applications.

Messaging mechanism

On top of Mailbox/MU drivers, a messaging system can be implemented, allowing messages to send between multiple endpoints created on each of the CPUs. The RPMsg-Lite library of the MCSDK provides this ability and serves as the preferred MCUXpresso SDK messaging library. It implements ring buffers in shared memory for messages exchange without the need of a locking mechanism.

The RPMsg-Lite provides the abstraction layer and can be easily ported to different multicore platforms and environments (Operating Systems). The advantages of such a messaging system are ease of use (there is no need to study behavior of the used underlying hardware) and smooth application code portability between platforms due to unified messaging API.

However, this costs several kB of code and data memory. The MCUXpresso SDK is accompanied by the RPMsg-Lite documentation and several multicore examples. You can also obtain the latest RPMsg-Lite code from the GitHub account github.com/nxp-mcuxpresso/rpmsg-lite.

Remote procedure calls

To facilitate the IPC even more and to allow the remote functions invocation, the remote procedure call mechanism can be implemented. The eRPC of the MCSDK serves for these purposes and allows the ability to invoke a software routine on a remote system via a simple local function call. Utilizing different transport layers, it is possible to communicate between individual cores of multicore SoCs (via RPMsg-Lite) or between separate processors (via SPI, UART, or TCP/IP). The eRPC is mostly applicable to the MPU parts with enough of memory resources like i.MX parts.

The eRPC library allows you to export existing C functions without having to change their prototypes (in most cases). It is accompanied by the code generator tool that generates the shim code for serialization and invocation based on the IDL file with definitions of data types and remote interfaces (API).

If the communicating peer is running as a Linux OS user-space application, the generated code can be either in C/C++ or Python.

Using the eRPC simplifies the access to services implemented on individual cores. This way, the following types of applications running on dedicated cores can be easily interfaced:

- Communication stacks (USB, Thread, Bluetooth Low Energy, Zigbee)
- Sensor aggregation/fusion applications
- Encryption algorithms
- Virtual peripherals

The eRPC is publicly available from the following GitHub account: github.com/EmbeddedRPC/erpc. Also, the MCUXpresso SDK is accompanied by the eRPC code and several multicore and multiprocessor eRPC examples.

The mentioned IPC levels demonstrate the scalability of the Multicore SDK library. Based on application needs, different IPC techniques can be used. It depends on the complexity, required speed, memory resources, system design, and so on. The MCSDK brings users the possibility for quick and easy development of multicore and multiprocessor applications.

Changelog Multicore SDK

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[25.12.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.14.0
 - eRPC generator (erpcgen) v1.14.0
 - Multicore Manager (MCMgr) v5.0.2
 - RPMsg-Lite v5.3.0

[25.09.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.14.0

- eRPC generator (erpcgen) v1.14.0
- Multicore Manager (MCMgr) v5.0.1
- RPPMsg-Lite v5.2.1

[25.06.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.14.0
 - eRPC generator (erpcgen) v1.14.0
 - Multicore Manager (MCMgr) v5.0.0
 - RPPMsg-Lite v5.2.0

[25.03.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.7
 - RPPMsg-Lite v5.1.4

[24.12.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.6
 - RPPMsg-Lite v5.1.3

[2.16.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.2

[2.15.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.12.0
 - eRPC generator (erpcgen) v1.12.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.1

[2.14.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.11.0
 - eRPC generator (erpcgen) v1.11.0
 - Multicore Manager (MCMgr) v4.1.4
 - RPSMsg-Lite v5.1.0

[2.13.0_imxrt1180a0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPSMsg-Lite v5.0.0

[2.13.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPSMsg-Lite v5.0.0

[2.12.0_imx93]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.1

[2.12.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.0

[2.11.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.1

[2.11.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.0

[2.10.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.1
 - eRPC generator (erpcgen) v1.8.1
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.2

[2.9.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.0
 - eRPC generator (erpcgen) v1.8.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.1

[2.8.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.4
 - eRPC generator (erpcgen) v1.7.4
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.1.0

[2.7.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.3
 - eRPC generator (erpcgen) v1.7.3
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.0.0

[2.6.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.2
 - eRPC generator (erpcgen) v1.7.2
 - Multicore Manager (MCMgr) v4.0.3
 - RPSMsg-Lite v2.2.0

[2.5.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.1
 - eRPC generator (erpcgen) v1.7.1
 - Multicore Manager (MCMgr) v4.0.2
 - RPSMsg-Lite v2.0.2

[2.4.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.0
 - eRPC generator (erpcgen) v1.7.0
 - Multicore Manager (MCMgr) v4.0.1
 - RPSMsg-Lite v2.0.1

[2.3.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.6.0
 - eRPC generator (erpcgen) v1.6.0
 - Multicore Manager (MCMgr) v4.0.0
 - RPSMsg-Lite v1.2.0

[2.3.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.5.0
 - eRPC generator (erpcgen) v1.5.0
 - Multicore Manager (MCMgr) v3.0.0
 - RPSMsg-Lite v1.2.0

[2.2.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.4.0
 - eRPC generator (erpcgen) v1.4.0
 - Multicore Manager (MCMgr) v2.0.1
 - RPSMsg-Lite v1.1.0

[2.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.3.0
 - eRPC generator (erpcgen) v1.3.0

[2.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.2.0
 - eRPC generator (erpcgen) v1.2.0
 - Multicore Manager (MCMgr) v2.0.0
 - RPSMsg-Lite v1.0.0

[1.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.1.0
 - Multicore Manager (MCMgr) v1.1.0
 - Open-AMP / RPSMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev01

[1.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.0.0
 - Multicore Manager (MCMgr) v1.0.0
 - Open-AMP / RPSMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev00

Multicore SDK Components

RPMSG-Lite

MCUXpresso SDK : mcuxsdk-middleware-rpmsg-lite

Overview This repository is for MCUXpresso SDK RPMSG-Lite middleware delivery and it contains RPMSG-Lite component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run RPMSG-Lite examples that are based on mcux-sdk-middleware-rpmsg-lite component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

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

For Further API documentation, please look at [doxygen documentation](#)

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the rpmsg-lite project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

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

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

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

For overview please read RPMSG-Lite VirtIO Overview.

For RPMSG-Lite Design Considerations please read RPMSG-Lite Design Considerations.

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

RPMsg protocol was not documented, and its only definition was given by the Linux Kernel and legacy OpenAMP implementations. This has changed with [1] which is a standardization protocol allowing multiple different implementations to coexist and still be mutually compatible.

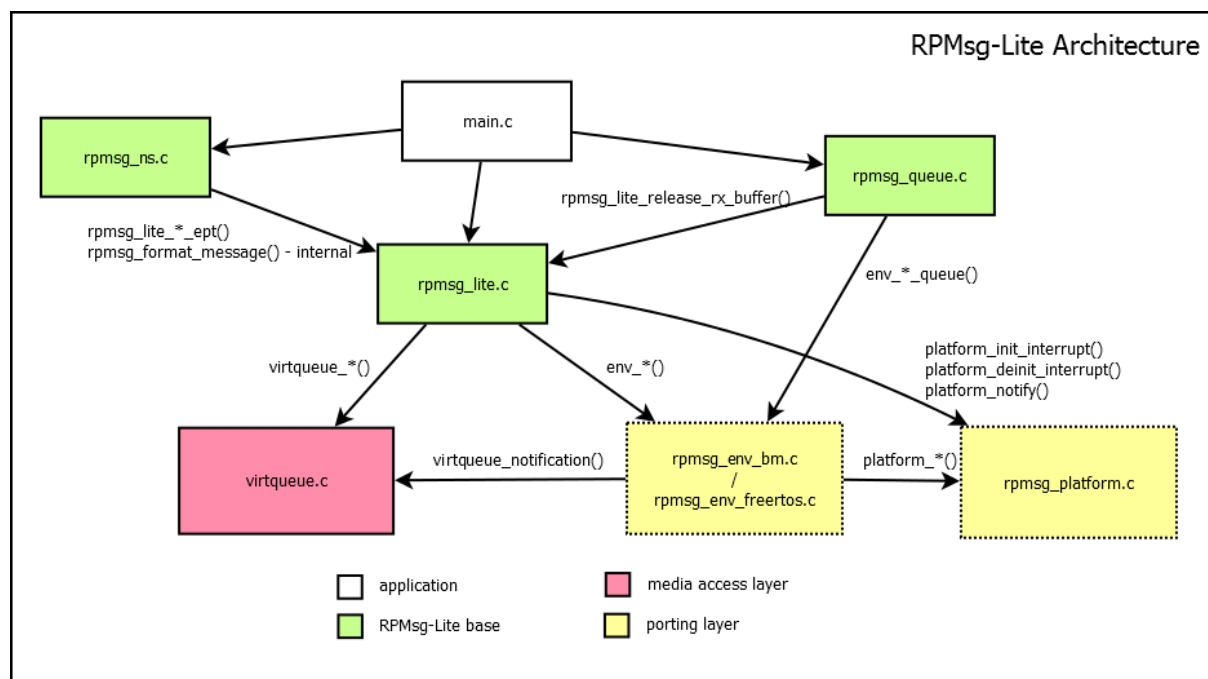
Small MCU-based systems often do not implement dynamic memory allocation. The creation of static API in RPMsg-Lite enables another reduction of resource usage. Not only does the dynamic allocation adds another 5 KB of code size, but also communication is slower and less deterministic, which is a property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

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

Implementation The implementation of RPMsg-Lite can be divided into three sub-components, from which two are optional. The core component is situated in `rpmsg_lite.c`. Two optional components are used to implement a blocking receive API (in `rpmsg_queue.c`) and dynamic “named” endpoint creation and deletion announcement service (in `rpmsg_ns.c`).

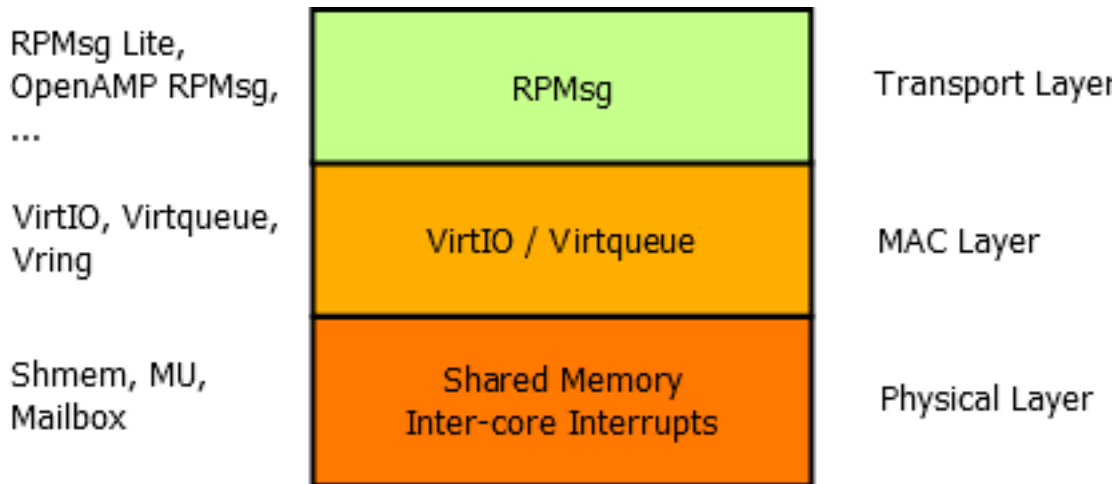
The actual “media access” layer is implemented in `virtqueue.c`, which is one of the few files shared with the OpenAMP implementation. This layer mainly defines the shared memory model, and internally defines used components such as `vring` or `virtqueue`.

The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. (The bare metal environment already exists and is implemented in `rpmsg_env_bm.c`, and the FreeRTOS environment is implemented in `rpmsg_env_freertos.c` etc.) Only the source file, which matches the used environment, is included in the target application project. The second sublayer is implemented in `rpmsg_platform.c` and defines low-level functions for interrupt enabling, disabling, and triggering mainly. The situation is described in the following figure:



RPMsg-Lite core sub-component This subcomponent implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer. This is realized by using so-called endpoints. Each endpoint can be assigned a different receive callback function.

However, it is important to notice that the callback is executed in an interrupt environment in current design. Therefore, certain actions like memory allocation are discouraged to execute in the callback. The following figure shows the role of RPMsg in an ISO/OSI-like layered model:



Queue sub-component (optional) This subcomponent is optional and requires implementation of the `env_*_queue()` functions in the environment porting layer. It uses a blocking receive API, which is common in RTOS-environments. It supports both copy and nocopy blocking receive functions.

Name Service sub-component (optional) This subcomponent is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about “named” endpoint (in other words, channel) creation or deletion and to receive these announcement taking any user-defined action in an application callback. The endpoint address used to receive name service announcements is arbitrarily fixed to be 53 (0x35).

Usage The application should put the `/rpmmsg_lite/lib/include` directory to the include path and in the application, include either the `rpmmsg_lite.h` header file, or optionally also include the `rpmmsg_queue.h` and/or `rpmmsg_ns.h` files. Both porting sublayers should be provided for you by NXP, but if you plan to use your own RTOS, all you need to do is to implement your own environment layer (in other words, `rpmmsg_env_myrtos.c`) and to include it in the project build.

The initialization of the stack is done by calling the `rpmmsg_lite_master_init()` on the master side and the `rpmmsg_lite_remote_init()` on the remote side. This initialization function must be called prior to any RPMsg-Lite API call. After the init, it is wise to create a communication endpoint, otherwise communication is not possible. This can be done by calling the `rpmmsg_lite_create_ept()` function. It optionally accepts a last argument, where an internal context of the endpoint is created, just in case the `RL_USE_STATIC_API` option is set to 1. If not, the stack internally calls `env_alloc()` to allocate dynamic memory for it. In case a callback-based receiving is to be used, an ISR-callback is registered to each new endpoint with user-defined callback data pointer. If a blocking receive is desired (in case of RTOS environment), the `rpmmsg_queue_create()` function must be called before calling `rpmmsg_lite_create_ept()`. The queue handle is passed to the endpoint creation function as a callback data argument and the callback function is set to `rpmmsg_queue_rx_cb()`. Then, it is possible to use `rpmmsg_queue_receive()` function to listen on a queue object for incoming messages. The `rpmmsg_lite_send()` function is used to send messages to the other side.

The RPMsg-Lite also implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

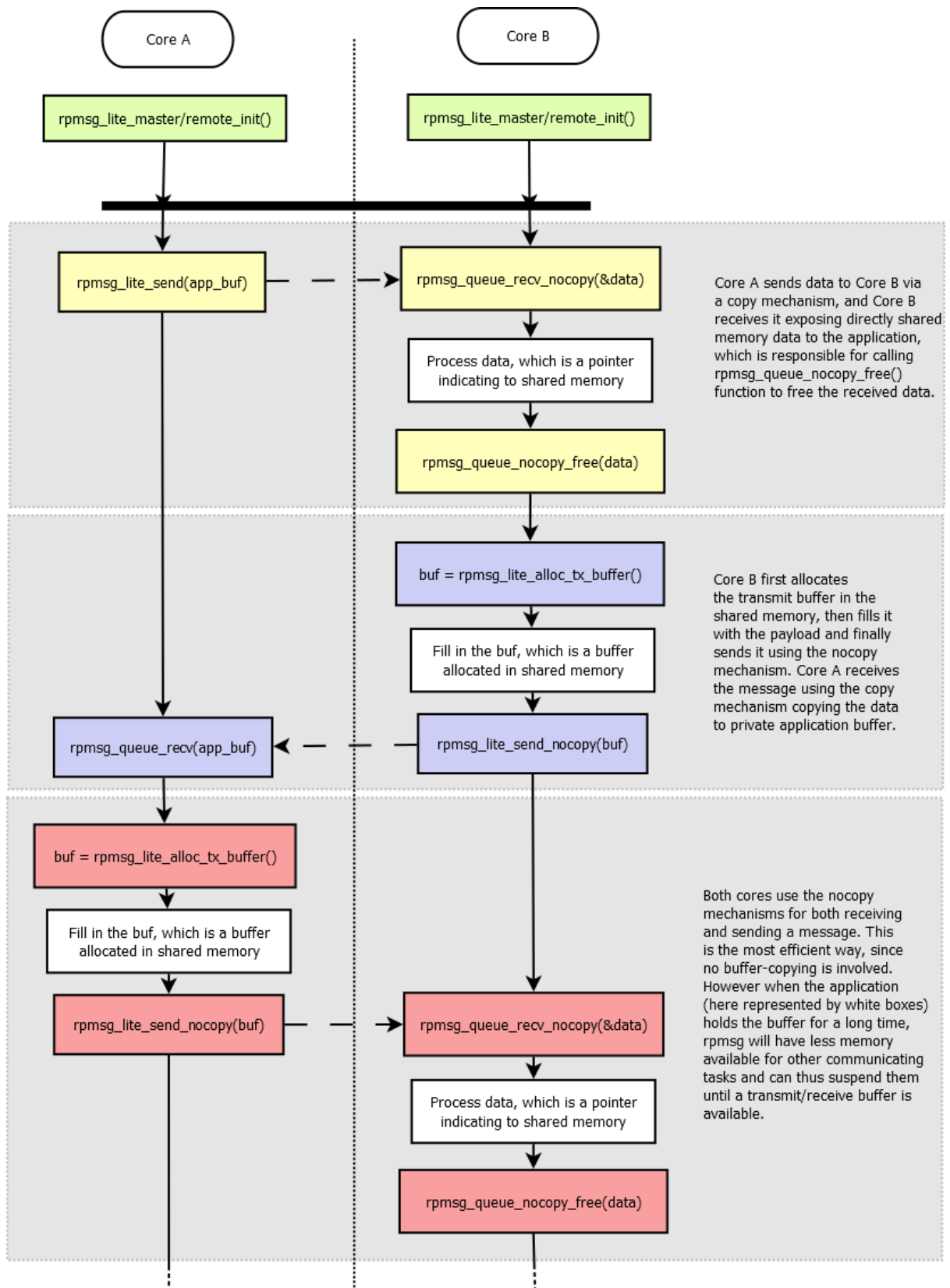
no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the `rpmmsg_lite_alloc_tx_buffer()` function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the `rpmmsg_lite_alloc_tx_buffer()` size output parameter).
- Call the `rpmmsg_lite_send_nocopy()` function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the `rpmmsg_lite_send_nocopy()` function description below.

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

- Call the `rpmmsg_queue_rcv_nocopy()` function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the `rpmmsg_queue_nocopy_free()` function to release the virtio buffer and to make it available for the next data transfer.

The user is responsible for destroying any RPMsg-Lite objects he has created in case of deinitialization. In order to do this, the function `rpmmsg_queue_destroy()` is used to destroy a queue, `rpmmsg_lite_destroy_ept()` is used to destroy an endpoint and finally, `rpmmsg_lite_deinit()` is used to deinitialize the RPMsg-Lite intercore communication stack. Deinitialize all endpoints using a queue before deinitializing the queue. Otherwise, you are actively invalidating the used queue handle, which is not allowed. RPMsg-Lite does not check this internally, since its main aim is to be lightweight.



Examples RPMsg Lite multicore examples are part of NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages. To get the board list with multicore support (RPMsg Lite included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded,

see

`<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples` for RPMsg_Lite multicore examples with 'rpsmsg_lite_' name prefix.

Another way of getting NXP MCUXpressoSDK RPMsg_Lite multicore examples is using the [mcuxsdk-manifests](#) Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk-manifests repo in [readme section](#). Once done the armgcc rpsmsg_lite examples can be found in

`mcuxsdk/examples/_<board_name>/multicore_examples`

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the RPMsg_Lite examples use the 'rpsmsg_lite_' name prefix.

Notes

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

- `rpsmsg_env_bm.c`
- `rpsmsg_env_freertos.c`
- `rpsmsg_env_xos.c`
- `rpsmsg_env_threadx.c`

The rest of environment layers has been created and used in some experimental projects, it has been running well at the time of creation but due to the lack of unit testing there is no guarantee it is still fully functional.

Shared memory configuration It is important to correctly initialize/configure the shared memory for data exchange in the application. The shared memory must be accessible from both the master and the remote core and it needs to be configured as Non-Cacheable memory. Dedicated shared memory section in linker file is also a good practise, it is recommended to use linker files from MCUXpressoSDK packages for NXP devices based applications. It needs to be ensured no other application part/component is unintentionally accessing this part of memory.

Configuration options The RPMsg-Lite can be configured at the compile time. The default configuration is defined in the `rpsmsg_default_config.h` header file. This configuration can be customized by the user by including `rpsmsg_config.h` file with custom settings. The following table summarizes all possible RPMsg-Lite configuration options.

Config- uration option	De- fault value	Usage
RL_MS_PE (1)		Delay in milliseconds used in non-blocking API functions for polling.
RL_BUFFE (496)		Size of the buffer payload, it must be more than 1 byte, and has to be word align (including rpmsg header size 16 bytes), if not it will be aligned up
RL_BUFFE (2)		Number of the buffers, it must be power of two (2, 4, ...)
RL_API_H (1)		Zero-copy API functions enabled/disabled.
RL_USE_S' (0)		Static API functions (no dynamic allocation) enabled/disabled.
RL_USE_D (0)		Memory cache management of shared memory. Use in case of data cache is enabled for shared memory.
RL_CLEAF (0)		Clearing used buffers before returning back to the pool of free buffers enabled/disabled.
RL_USE_M (0)		When enabled IPC interrupts are managed by the Multicore Manager (IPC interrupts router), when disabled RPMsg-Lite manages IPC interrupts by itself.
RL_USE_E (0)		When enabled the environment layer uses its own context. Required for some environments (QNX). The default value is 0 (no context, saves some RAM).
RL_DEBU (0)		When enabled buffer pointers passed to <code>rpmsg_lite_send_nocopy()</code> and <code>rpmsg_lite_release_rx_buffer()</code> functions (enabled by <code>RL_API_HAS_ZEROCOPY</code> config) are checked to avoid passing invalid buffer pointer. The default value is 0 (disabled). Do not use in RPMsg-Lite to Linux configuration.
RL_ALLOV (0)		When enabled the opposite side is notified each time received buffers are consumed and put into the queue of available buffers. Enable this option in RPMsg-Lite to Linux configuration to allow unblocking of the Linux blocking send. The default value is 0 (RPMsg-Lite to RPMsg-Lite communication).
RL_ALLOV (0)		It allows to define custom shared memory configuration and replacing the shared memory related global settings from <code>rpmsg_config.h</code> . This is useful when multiple instances are running in parallel but different shared memory arrangement (vring size & alignment, buffers size & count) is required. The default value is 0 (all RPMsg_Lite instances use the same shared memory arrangement as defined by common config macros).
RL_ASSER	see rpmsg	Assert implementation.

How to format rpmsg-lite code To format code, use the application developed by Google, named *clang-format*. This tool is part of the [llvm](#) project. Currently, the clang-format 10.0.0 version is used for rpmsg-lite. The set of style settings used for clang-format is defined in the `.clang-format` file, placed in a root of the rpmsg-lite directory where Python script `run_clang_format.py` can be executed. This script executes the application named *clang-format.exe*. You need to have the path of this application in the OS's environment path, or you need to change the script.

References

[1] M. Novak, M. Cingel, **Lockless Shared Memory Based Multicore Communication Protocol**
Copyright © 2016 Freescale Semiconductor, Inc. Copyright © 2016-2025 NXP

Changelog RPMSG-Lite All notable changes to this project will be documented in this file. The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[v5.3.0]

Added

- RT700 porting layer added support to send rpmsg messages between CM33_0 <-> Hifi1 and CM33_1 <-> Hifi4 cores.
- Add new platform macro `RL_PLATFORM_MAX_ISR_COUNT` this will set number of IRQ count per platform. This macro is then used in environment layers to set `isr_table` size where irq handles are registered. It size should match the bit length of `VQ_ID` so all combinations can fit into table.
- Unit tests updated to improve code coverage, new unit tests added covering static allocations in rtos environment layers.

Fixed

- `virtio.h` removed `typedef uint8_t boolean` and in its place use standard C99 `bool` type to avoid potential type conflicts.
- `env_acquire_sync_lock()` and `env_release_sync_lock()` synchronization primitives removed
- Kconfig consolidation, when `RL_ALLOW_CUSTOM_SHMEM_CONFIG` enabled the `platform_get_custom_shmem_config()` function needs to be implemented in platform layer to provide custom shared memory configuration for RPMsg-Lite instance.

v5.2.1

Added

- Doc added RPMSG-Lite VirtIO Overview
- Doc added RPMSG-Lite Design Considerations
- Added `frdmimxrt1186` unit testing

Changed

- Remove limitation that `RL_BUFFER_SIZE` needs to be power of 2. It just has to be more than 16 bytes, e.g. 16 bytes of rpmsg header and payload size at least 1 byte and word aligned, if not it will be aligned up.

Fixed

- Fixed CERT-C INT31-C violation in `platform_notify` function in `rpmsg_platform.c` for `imxrt700_m33`, `imxrt700_hifi4`, `imxrt700_hifi1` platforms

v5.2.0

Added

- Add MCXL20 porting layer and unit testing
- New utility macro `RL_CALCULATE_BUFFER_COUNT_DOWN_SAFE` to safely determine maximum buffer count within shared memory while preventing integer underflow.
- RT700 platform add support for MCMGR in DSPs

Changed

- Change `rpmsg_platform.c` to support new MCMGR API
- Improved input validation in initialization functions to properly handle insufficient memory size conditions.
- Refactored repeated buffer count calculation pattern for better code maintainability.
- To make sure that remote has already registered IRQ there is required App level IPC mechanism to notify master about it

Fixed

- Fixed `env_wait_for_link_up` function to handle timeout in link state checks for baremetal and qnx environment, `RL_BLOCK` mode can be used to wait indefinitely.
- Fixed CERT-C INT31-C violation by adding compile-time check to ensure `RL_PLATFORM_HIGHEST_LINK_ID` remains within safe range for 16-bit casting in virtqueue ID creation.
- Fixed CERT-C INT30-C violations by adding protection against unsigned integer underflow in shared memory calculations, specifically in `shmem_length - (uint32_t)RL_VRING_OVERHEAD` and `shmem_length - 2U * shmem_config.vring_size` expressions.
- Fixed CERT INT31-C violation in `platform_interrupt_disable()` and similar functions by replacing unsafe cast from `uint32_t` to `int32_t` with a return of 0 constant.
- Fixed unsigned integer underflow in `rpmsg_lite_alloc_tx_buffer()` where subtracting header size from buffer size could wrap around if buffer was too small, potentially leading to incorrect buffer sizing.
- Fixed CERT-C INT31-C violation in `rpmsg_lite.c` where `size` parameter was cast from `uint32_t` to `uint16_t` without proper validation.
 - Applied consistent masking approach to both `size` and `flags` parameters: `(uint16_t)(value & 0xFFFFU)`.
 - This fix prevents potential data loss when `size` values exceed 65535.
- Fixed CERT INT31-C violation in `env_memset` functions by explicitly converting `int32_t` values to unsigned char using bit masking. This prevents potential data loss or misinterpretation when passing values outside the unsigned char range (0-255) to the standard `memset()` function.
- Fixed CERT-C INT31-C violations in RPMsg-Lite environment porting: Added validation checks for signed-to-unsigned integer conversions to prevent data loss and misinterpretation.
 - `rpmsg_env_freertos.c`: Added validation before converting `int32_t` to `UBaseType_t`.
 - `rpmsg_env_qnx.c`: Fixed format string and added validation before assigning to `mqstat` fields.
 - `rpmsg_env_threadx.c`: Added validation to prevent integer overflow and negative values.
 - `rpmsg_env_xos.c`: Added range checking before casting to `uint16_t`.
 - `rpmsg_env_zephyr.c`: Added validation before passing values to `k_msgq_init`.
- Fixed a CERT INT31-C compliance issue in `env_get_current_queue_size()` function where an unsigned queue count was cast to a signed `int32_t` without proper validation, which could lead to lost or misinterpreted data if queue size exceeded `INT32_MAX`.
- Fixed CERT INT31-C violation in `rpmsg_platform.c` where `memcmp()` return value (signed int) was compared with unsigned constant without proper type handling.

- Fixed CERT INT31-C violation in `rpmsg_platform.c` where casting from `uint32_t` to `uint16_t` could potentially result in data loss. Changed length variable type from `uint16_t` to `uint32_t` to properly handle memory address differences without truncation.
- Fixed potential integer overflow in `env_sleep_msec()` function in ThreadX environment implementation by rearranging calculation order in the sleep duration formula.
- Fixed CERT-C INT31-C violation in RPMsg-Lite where bitwise NOT operations on integer constants were performed in signed integer context before being cast to unsigned. This could potentially lead to misinterpreted data on `imx943` platform.
- Added `RL_MAX_BUFFER_COUNT` (32768U) and `RL_MAX_VRING_ALIGN` (65536U) limit to ensure alignment values cannot contribute to integer overflow
- Fixed CERT INT31-C violation in `vring_need_event()`, added cast to `uint16_t` for each operand.

v5.1.4 - 27-Mar-2025

Added

- Add KW43B43 porting layer

Changed

- Doxygen bump to version 1.9.6

v5.1.3 - 13-Jan-2025

Added

- Memory cache management of shared memory. Enable with `#define RL_USE_DCACHE (1)` in `rpmsg_config.h` in case of data cache is used.
- Cmake/Kconfig support added.
- Porting layers for `imx95`, `imxrt700`, `mcmxw71x`, `mcmxw72x`, `kw47b42` added.

v5.1.2 - 08-Jul-2024

Changed

- Zephyr-related changes.
- Minor Misra corrections.

v5.1.1 - 19-Jan-2024

Added

- Test suite provided.
- Zephyr support added.

Changed

- Minor changes in platform and env. layers, minor test code updates.

v5.1.0 - 02-Aug-2023**Added**

- RPMMsg-Lite: Added aarch64 support.

Changed

- RPMMsg-Lite: Increased the queue size to (2 * RL_BUFFER_COUNT) to cover zero copy cases.
- Code formatting using LLVM16.

Fixed

- Resolved issues in ThreadX env. layer implementation.

v5.0.0 - 19-Jan-2023**Added**

- Timeout parameter added to `rpmmsg_lite_wait_for_link_up` API function.

Changed

- Improved debug check buffers implementation - instead of checking the pointer fits into shared memory check the presence in the VirtIO ring descriptors list.
- VRING_SIZE is set based on number of used buffers now (as calculated in `vring_init`) - updated for all platforms that are not communicating to Linux rpmmsg counterpart.

Fixed

- Fixed wrong RL_VRING_OVERHEAD macro comment in platform.h files
- Misra corrections.

v4.0.0 - 20-Jun-2022**Added**

- Added support for custom shared memory arrangement per the RPMMsg_Lite instance.
- Introduced new `rpmmsg_lite_wait_for_link_up()` API function - this allows to avoid using busy loops in rtos environments, GitHub PR [#21](#).

Changed

- Adjusted `rpmmsg_lite_is_link_up()` to return RL_TRUE/RL_FALSE.

v3.2.0 - 17-Jan-2022

Added

- Added support for i.MX8 MP multicore platform.

Changed

- Improved static allocations - allow OS-specific objects being allocated statically, GitHub PR [#14](#).
- Aligned rpmsg_env_xos.c and some platform layers to latest static allocation support.

Fixed

- Minor Misra and typo corrections, GitHub PR [#19](#), [#20](#).

v3.1.2 - 16-Jul-2021

Added

- Addressed MISRA 21.6 rule violation in rpmsg_env.h (use SDK's PRINTF in MCUXpressoSDK examples, otherwise stdio printf is used).
- Added environment layers for XOS.
- Added support for i.MX RT500, i.MX RT1160 and i.MX RT1170 multicore platforms.

Fixed

- Fixed incorrect description of the rpmsg_lite_get_endpoint_from_addr function.

Changed

- Updated RL_BUFFER_COUNT documentation (issue [#10](#)).
- Updated imxrt600_hifi4 platform layer.

v3.1.1 - 15-Jan-2021

Added

- Introduced RL_ALLOW_CONSUMED_BUFFERS_NOTIFICATION config option to allow opposite side notification sending each time received buffers are consumed and put into the queue of available buffers.
- Added environment layers for Threadx.
- Added support for i.MX8QM multicore platform.

Changed

- Several MISRA C-2012 violations addressed.

v3.1.0 - 22-Jul-2020

Added

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed (7.4).
- Fixed missing lock in `rpmsg_lite_rx_callback()` for QNX env.
- Correction of `rpmsg_lite_instance` structure members description.
- Address -Waddress-of-packed-member warnings in GCC9.

Changed

- Clang update to v10.0.0, code re-formatted.

v3.0.0 - 20-Dec-2019**Added**

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed, incl. data types consolidation.
- Code formatted.

v2.2.0 - 20-Mar-2019**Added**

- Added configuration macro `RL_DEBUG_CHECK_BUFFERS`.
- Several MISRA violations fixed.
- Added environment layers for QNX and Zephyr.
- Allow environment context required for some environment (controlled by the `RL_USE_ENVIRONMENT_CONTEXT` configuration macro).
- Data types consolidation.

v1.1.0 - 28-Apr-2017**Added**

- Supporting i.MX6SX and i.MX7D MPU platforms.
- Supporting LPC5411x MCU platform.
- Baremetal and FreeRTOS support.
- Support of copy and zero-copy transfer.
- Support of static API (without dynamic allocations).

Multicore Manager

MCUXpresso SDK : mcuxsdk-middleware-mcmgr (Multicore Manager)

Overview This repository is for MCUXpresso SDK Multicore Manager middleware delivery and it contains Multicore Manager component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run Multicore Manager examples that are based on mcux-sdk-middleware-mcmgr component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

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

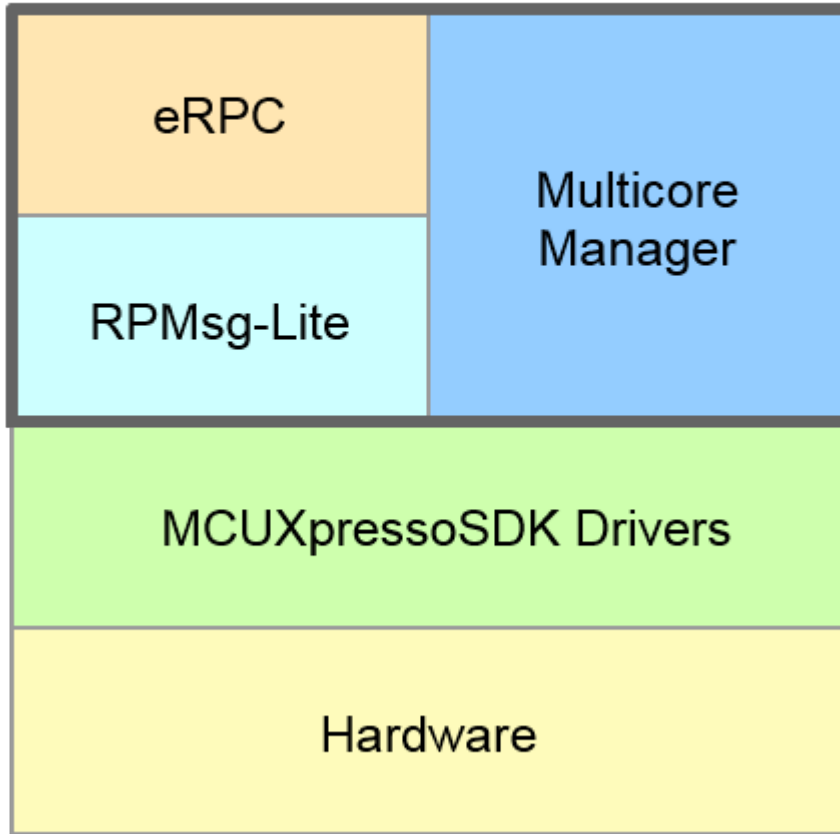
For Further API documentation, please look at [doxygen documentation](#)

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the mcmgr project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems. This library is distributed as a part of the Multicore SDK (MCSDK). Together, the MCSDK and the MCUXpresso SDK (SDK) form a framework for development of software for NXP multicore devices.

The MCMGR component is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr` directory.



The Multicore Manager provides the following major functions:

- Maintains information about all cores in system.
- Secondary/auxiliary core(s) startup and shutdown.
- Remote core monitoring and event handling.

Usage of the MCMGR software component The main use case of MCMGR is the secondary/auxiliary core start. This functionality is performed by the public API function.

Example of MCMGR usage to start secondary core:

```
#include "mcmgr.h"

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

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
}
```

(continues on next page)

(continued from previous page)

```

/* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass "1" as startup data,
↳ starting synchronously. */
MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 1, kMCMGR_Start_Synchronous);
.
.
.
/* Stop secondary core execution. */
MCMGR_StopCore(kMCMGR_Core1);
}

```

Some platforms allow stopping and re-starting the secondary core application again, using the MCMGR_StopCore / MCMGR_StartCore API calls. It is necessary to ensure the initially loaded image is not corrupted before re-starting, especially if it deals with the RAM target. Cache coherence has to be considered/ensured as well.

It could also happen that the secondary core application stops running correctly and the primary core application does not know about that situation. Therefore, it is beneficial to implement a mechanism for core health monitoring. The *test_heartbeat* unit test can serve as an example how to ensure that: secondary core could periodically send heartbeat signals to the primary core using MCMGR_TriggerEvent() API to indicate that it is alive and functioning properly.

Another important MCMGR feature is the ability for remote core monitoring and handling of events such as reset, exception, and application events. Application-specific callback functions for events are registered by the MCMGR_RegisterEvent() API. Triggering these events is done using the MCMGR_TriggerEvent() API. mcmgr_event_type_t enums all possible event types.

An example of MCMGR usage for remote core monitoring and event handling. Code for the primary side:

```

#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA (1)
#define APP_NUMBER_OF_CORES (2)
#define APP_SECONDARY_CORE kMCMGR_Core1

/* Callback function registered via the MCMGR_RegisterEvent() and triggered by MCMGR_TriggerEvent()
↳ called on the secondary core side */
void RPMsgRemoteReadyEventHandler(mcmgr_core_t coreNum, uint16_t eventData, void *context)
{
    uint16_t *data = &((uint16_t *)context)[coreNum];

    *data = eventData;
}

void main()
{
    uint16_t RPMsgRemoteReadyEventData[NUMBER_OF_CORES] = {0};

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

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Register the application event before starting the secondary core */
    MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent, RPMsgRemoteReadyEventHandler, (void
↳ *)RPMsgRemoteReadyEventData);

```

(continues on next page)

(continued from previous page)

```

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

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

```

Code for the secondary side:

```

#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA (1)

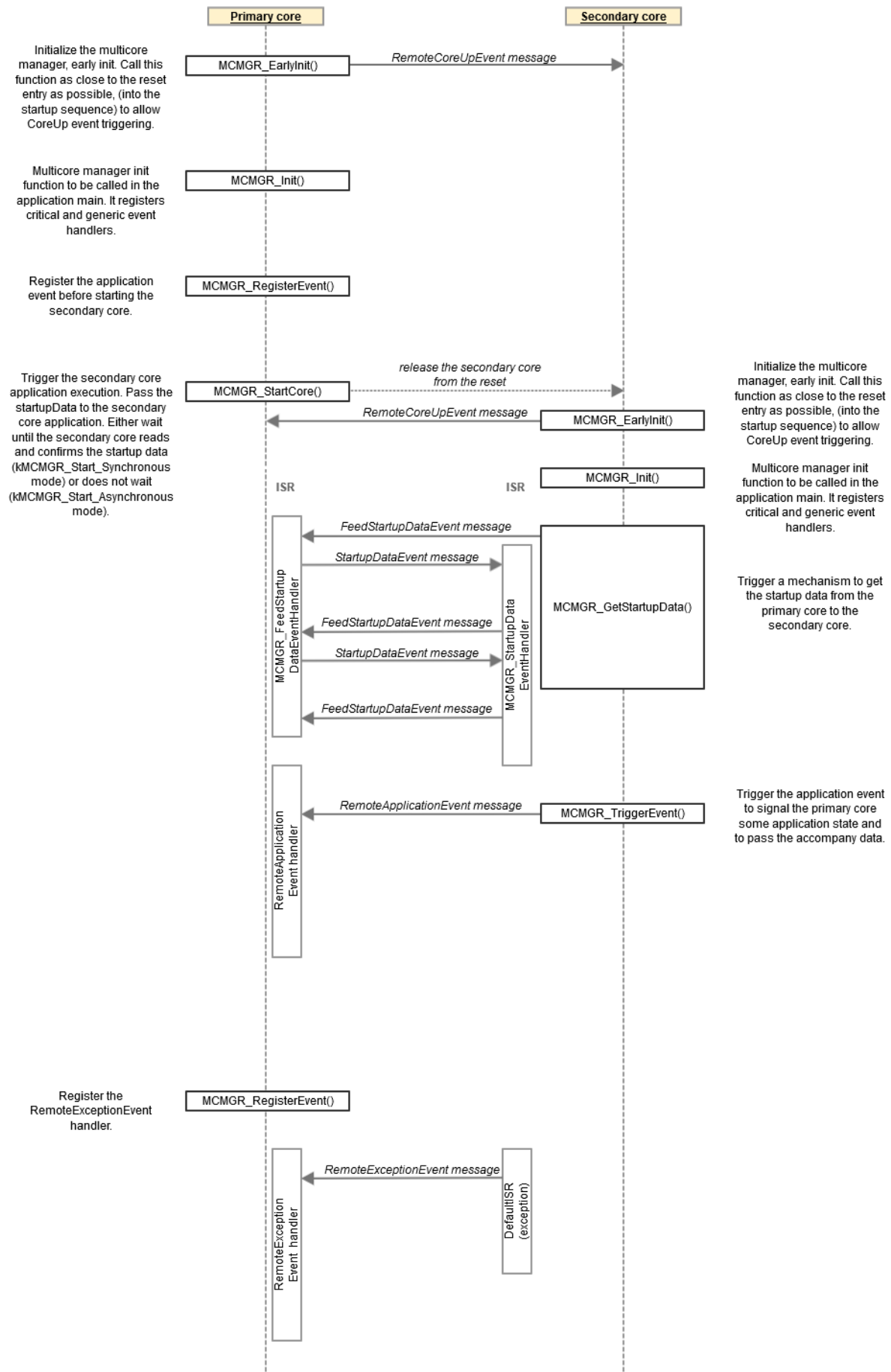
void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
    .
    .
    .

    /* Signal the to other core that we are ready by triggering the event and passing the APP_RPMSG_
↳READY_EVENT_DATA */
    MCMGR_TriggerEvent(kMCMGR_Core0, kMCMGR_RemoteApplicationEvent, APP_RPMSG_
↳READY_EVENT_DATA);
    .
    .
    .
}

```

MCMGR Data Exchange Diagram The following picture shows how the handshakes are supposed to work between the two cores in the MCMGR software.



Changelog Multicore Manager All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[v5.0.2]

Added

- Added gcov options and configs to support mcmgr code coverage
- Added new test_weak_mu_isr testcase for devices with MU peripheral
- Added new test_heartbeat testcase showing heartbeat mechanism between primary and secondary cores using the MCMGR

v5.0.1

Added

- Added frdmimxrt1186 unit testing

Changed

- [KW43] Rename core#1 reset control register

Fixed

- Added CX flag into CMakeLists.txt to allow c++ build compatibility.
- Fix path to mcmgr headers directory in doxyfile

v5.0.0

Added

- Added MCMGR_BUSY_POLL_COUNT macro to prevent infinite polling loops in MCMGR operations.
- Implemented timeout mechanism for all polling loops in MCMGR code.
- Added support to handle more then two cores. Breaking API change by adding parameter coreNum specifying core number in functions bellow.
 - MCMGR_GetStartupData(uint32_t *startupData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEvent(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEventForce(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - typedef void (*mcmgr_event_callback_t)(uint16_t data, void *context, mcmgr_core_t coreNum);

When registering the event with function `MCMGR_RegisterEvent()` user now needs to provide `callbackData` pointer to array of elements per every core in system (see `README.md` for example). In case of systems with only two cores the `coreNum` in callback can be ignored as events can arrive only from one core. Please see Porting guide for more details: `Porting-GuideTo_v5.md`

- Updated all porting files to support new MCMGR API.
- Added new platform specific include file `mcmgr_platform.h`. It will contain common platform specific macros that can be then used in `mcmgr` and application. e.g. platform core count `MCMGR_CORECOUNT` 4.
- Move all header files to new `inc` directory.
- Added new platform-specific include files `inc/platform/<platform_name>/mcmgr_platform.h`.

Added

- Add MCXL20 porting layer and unit testing

v4.1.7

Fixed

- `mcmgr_stop_core_internal()` function now returns `kStatus_MCMGR_NotImplemented` status code instead of `kStatus_MCMGR_Success` when device does not support stop of secondary core. Ports affected: `kw32w1`, `kw45b41`, `kw45b42`, `mcxw716`, `mcxw727`.

[v4.1.6]

Added

- Multicore Manager moved to standalone repository.
- Add porting layers for `imxrt700`, `mcmxw727`, `kw47b42`.
- New `MCMGR_ProcessDeferredRxIsr()` API added.

[v4.1.5]

Added

- Add notification into `MCMGR_EarlyInit` and `mcmgr_early_init_internal` functions to avoid using uninitialized data in their implementations.

[v4.1.4]

Fixed

- Avoid calling tx isr callbacks when respective Messaging Unit Transmit Interrupt Enable flag is not set in the CR/TCR register.
- Messaging Unit RX and status registers are cleared after the initialization.

[v4.1.3]**Added**

- Add porting layers for imxrt1180.

Fixed

- mu_isr() updated to avoid calling tx isr callbacks when respective Transmit Interrupt Enable flag is not set in the CR/TCR register.
- mcmgr_mu_internal.c code adaptation to new supported SoCs.

[v4.1.2]**Fixed**

- Update mcmgr_stop_core_internal() implementations to set core state to kMCMGR_ResetCoreState.

[v4.1.0]**Fixed**

- Code adjustments to address MISRA C-2012 Rules

[v4.0.3]**Fixed**

- Documentation updated to describe handshaking in a graphic form.
- Minor code adjustments based on static analysis tool findings

[v4.0.2]**Fixed**

- Align porting layers to the updated MCUXpressoSDK feature files.

[v4.0.1]**Fixed**

- Code formatting, removed unused code

[v4.0.0]

Added

- Add new MCMGR_TriggerEventForce() API.

[v3.0.0]

Removed

- Removed MCMGR_LoadApp(), MCMGR_MapAddress() and MCMGR_SignalReady()

Modified

- Modified MCMGR_GetStartupData()

Added

- Added MCMGR_EarlyInit(), MCMGR_RegisterEvent() and MCMGR_TriggerEvent()
- Added the ability for remote core monitoring and event handling

[v2.0.1]

Fixed

- Updated to be Misra compliant.

[v2.0.0]

Added

- Support for lpcxpresso54114 board.

[v1.1.0]

Fixed

- Ported to KSDK 2.0.0.

[v1.0.0]

Added

- Initial release.

eRPC

MCUXpresso SDK : mcuxsdk-middleware-erpc

Overview This repository is for MCUXpresso SDK eRPC middleware delivery and it contains eRPC component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run eRPC examples that are based on mcux-sdk-middleware-erpc component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

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

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the eRPC project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

eRPC

- [MCUXpresso SDK : mcuxsdk-middleware-erpc](#)

- [Overview](#)
- [Documentation](#)
- [Setup](#)
- [Contribution](#)

- [eRPC](#)

- [About](#)
- [Releases](#)
 - * [Edge releases](#)
- [Documentation](#)
- [Examples](#)
- [References](#)
- [Directories](#)
- [Building and installing](#)
 - * [Requirements](#)
 - [Windows](#)
 - [Mac OS X](#)
 - * [Building](#)
 - [CMake and KConfig](#)
 - [Make](#)

** Installing for Python*

- *Known issues and limitations*
- *Code providing*

About

eRPC (Embedded RPC) is an open source Remote Procedure Call (RPC) system for multichip embedded systems and heterogeneous multicore SoCs.

Unlike other modern RPC systems, such as the excellent [Apache Thrift](#), eRPC distinguishes itself by being designed for tightly coupled systems, using plain C for remote functions, and having a small code size (<5kB). It is not intended for high performance distributed systems over a network.

eRPC does not force upon you any particular API style. It allows you to export existing C functions, without having to change their prototypes. (There are limits, of course.) And although the internal infrastructure is written in C++, most users will be able to use only the simple C setup APIs shown in the examples below.

A code generator tool called `erpcgen` is included. It accepts input IDL files, having an `.erpc` extension, that have definitions of your data types and remote interfaces, and generates the shim code that handles serialization and invocation. `erpcgen` can generate either C/C++ or Python code.

Example `.erpc` file:

```
// Define a data type.
enum LEDName { kRed, kGreen, kBlue }

// An interface is a logical grouping of functions.
interface IO {
    // Simple function declaration with an empty reply.
    set_led(LEDName whichLed, bool onOrOff) -> void
}
```

Client side usage:

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* init eRPC client IO service */
    initIO_client(client_manager);

    // Now we can call the remote function to turn on the green LED.
    set_led(kGreen, true);

    /* deinit objects */
    deinitIO_client();
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
}
```

(continues on next page)

(continued from previous page)

```

    erpc_transport_tcp_deinit(transport);
}

void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* scope for client service */
    {
        /* init eRPC client IO service */
        IO_client client(client_manager);

        // Now we can call the remote function to turn on the green LED.
        client.set_led(kGreen, true);
    }

    /* deinit objects */
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

Server side usage:

```

// Implement the remote function.
void set_led(LEDName whichLed, bool onOrOff) {
    // implementation goes here
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    erpc_service_t service = create_IO_service();

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, service);

    // Run the server.
    erpc_server_run();

    /* deinit objects */
    destroy_IO_service(service);
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

```

// Implement the remote function.
class IO : public IO_interface

```

(continues on next page)

(continued from previous page)

```
{
    /* eRPC call definition */
    void set_led(LEDName whichLed, bool onOrOff) override {
        // implementation goes here
    }
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    IO IOImpl;
    IO_service io(&IOImpl);

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, &io);

    /* poll for requests */
    erpc_status_t err = server.run();

    /* deinit objects */
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

A number of transports are supported, and new transport classes are easy to write.

Supported transports can be found in *erpc/erpc_c/transport* folder. E.g:

- CMSIS UART
- NXP Kinetis SPI and DSPI
- POSIX and Windows serial port
- TCP/IP (mostly for testing)
- [NXP RPMsg-Lite / RPMsg TTY](#)
- SPIdev Linux
- USB CDC
- NXP Messaging Unit

eRPC is available with an unrestrictive BSD 3-clause license. See the [LICENSE file](#) for the full license text.

Releases [eRPC releases](#)

Edge releases Edge releases can be found on [eRPC CircleCI](#) webpage. Choose build of interest, then platform target and choose ARTIFACTS tab. Here you can find binary application from chosen build.

Documentation Documentation is in the wiki section.

eRPC Infrastructure documentation

Examples Example IDL is available in the *examples/* folder.

Plenty of eRPC multicore and multiprocessor examples can be also found in NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages.

To get the board list with multicore support (eRPC included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded, see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for eRPC multicore examples (RPMsg_Lite or Messaging Unit transports used) or

<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples for eRPC multiprocessor examples (UART or SPI transports used).

eRPC examples use the 'erpc_' name prefix.

Another way of getting NXP MCUXpressoSDK eRPC multicore and multiprocessor examples is using the [mcux-sdk](#) Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk repo in [readme Overview section](#). Once done the armgcc eRPC examples can be found in

mcuxsdk/examples/<board_name>/multicore_examples or in

mcuxsdk/examples/<board_name>/multiprocessor_examples folders.

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the eRPC examples use the 'erpc_' name prefix.

References This section provides links to interesting erpc-based projects, articles, blogs or guides:

- [erpc \(EmbeddedRPC\) getting started notes](#)
- [ERPC Linux Local Environment Construction and Use](#)
- [The New Wio Terminal eRPC Firmware](#)

Directories *doc* - Documentation.

doxygen - Configuration and support files for running Doxygen over the eRPC C++ infrastructure and erpcgen code.

erpc_c - Holds C/C++ infrastructure for eRPC. This is the code you will include in your application.

erpc_python - Holds Python version of the eRPC infrastructure.

erpcgen - Holds source code for erpcgen and makefiles or project files to build erpcgen on Windows, Linux, and OS X.

erpcsniffer - Holds source code for erpcsniffer application.

examples - Several example IDL files.

mk - Contains common makefiles for building eRPC components.

test - Client/server tests. These tests verify the entire communications path from client to server and back.

utilities - Holds utilities which bring additional benefit to eRPC apps developers.

Building and installing These build instructions apply to host PCs and embedded Linux. For bare metal or RTOS embedded environments, you should copy the *erpc_c* directory into your application sources.

CMake and KConfig build:

It builds a static library of the eRPC C/C++ infrastructure, the `erpcgen` executable, and optionally the unit tests and examples.

CMake is compatible with gcc and clang. On Windows local MingGW downloaded by *script* can be used.

Make build:

It builds a static library of the eRPC C/C++ infrastructure, the `erpcgen` executable, and optionally the unit tests.

The makefiles are compatible with gcc or clang on Linux, OS X, and Cygwin. A Windows build of erpcgen using Visual Studio is also available in the *erpcgen/VisualStudio_v14* directory. There is also an Xcode project file in the *erpcgen* directory, which can be used to build erpcgen for OS X.

Requirements eRPC now support building **erpcgen**, **erpc_lib**, **tests** and **C examples** using CMake.

Requirements when using CMake:

- **CMake** (minimal version 3.20.0)
- Generator - **Make**, **Ninja**, ...
- **C/C++ compiler** - **GCC**, **CLANG**, ...
- **Binson** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Requirements when using Make:

- **Make**
- **C/C++ compiler - GCC, CLANG, ...**
- **Binson** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Windows Related steps to build **erpcgen** using **Visual Studio** are described in `erpcgen/VisualStudio_v14/readme_erpcgen.txt`.

To install MinGW, Bison, Flex locally on Windows:

```
./install_dependencies.ps1
* ^^^
#### Linux
^^^ bash
./install_dependencies.sh
```

Mandatory for case, when build for different architecture is needed

- gcc-multilib, g++-multilib

Mac OS X

```
./install_dependencies.sh
```

Building

CMake and KConfig eRPC use CMake and KConfig to configure and build eRPC related targets. KConfig can be edited by *prj.conf* or *menuconfig* when building.

Generate project, config and build. In *erpc/* execute:

```
cmake -B ./build # in erpc/build generate cmake project
cmake --build ./build --target menuconfig # Build menuconfig and configure erpcgen, erpc_lib, tests and
↳examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

****CMake will use the system's default compilers and generator**

If you want to use Windows and locally installed MinGW, use *CMake preset* :

```
cmake --preset mingw64 # Generate project in ./build using mingw64's make and compilers
cmake --build ./build --target menuconfig # Build menuconfig and configure erpcgen, erpc_lib, tests and
↳examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

Make To build the library and erpcgen, run from the repo root directory:

```
make
```

To install the library, erpcgen, and include files, run:

```
make install
```

You may need to sudo the make install.

By default this will install into */usr/local*. If you want to install elsewhere, set the *PREFIX* environment variable. Example for installing into */opt*:

```
make install PREFIX=/opt
```

List of top level Makefile targets:

- *erpc*: build the *liberpc.a* static library
- *erpcgen*: build the *erpcgen* tool
- *erpcsniffer*: build the sniffer tool
- *test*: build the unit tests under the *test* directory
- *all*: build all of the above
- *install*: install *liberpc.a*, *erpcgen*, and include files

eRPC code is validated with respect to the C++ 11 standard.

Installing for Python To install the Python infrastructure for eRPC see instructions in the *erpc python readme*.

Known issues and limitations

- Static allocations controlled by the `ERPC_ALLOCATION_POLICY` config macro are not fully supported yet, i.e. not all erpc objects can be allocated statically now. It deals with the ongoing process and the full static allocations support will be added in the future.

Code providing Repository on Github contains two main branches: **main** and **develop**. Code is developed on **develop** branch. Release version is created via merging **develop** branch into **main** branch.

Copyright 2014-2016 Freescale Semiconductor, Inc.

Copyright 2016-2025 NXP

eRPC Getting Started

Overview This *Getting Started User Guide* shows software developers how to use Remote Procedure Calls (RPC) in embedded multicore microcontrollers (eRPC).

The eRPC documentation is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

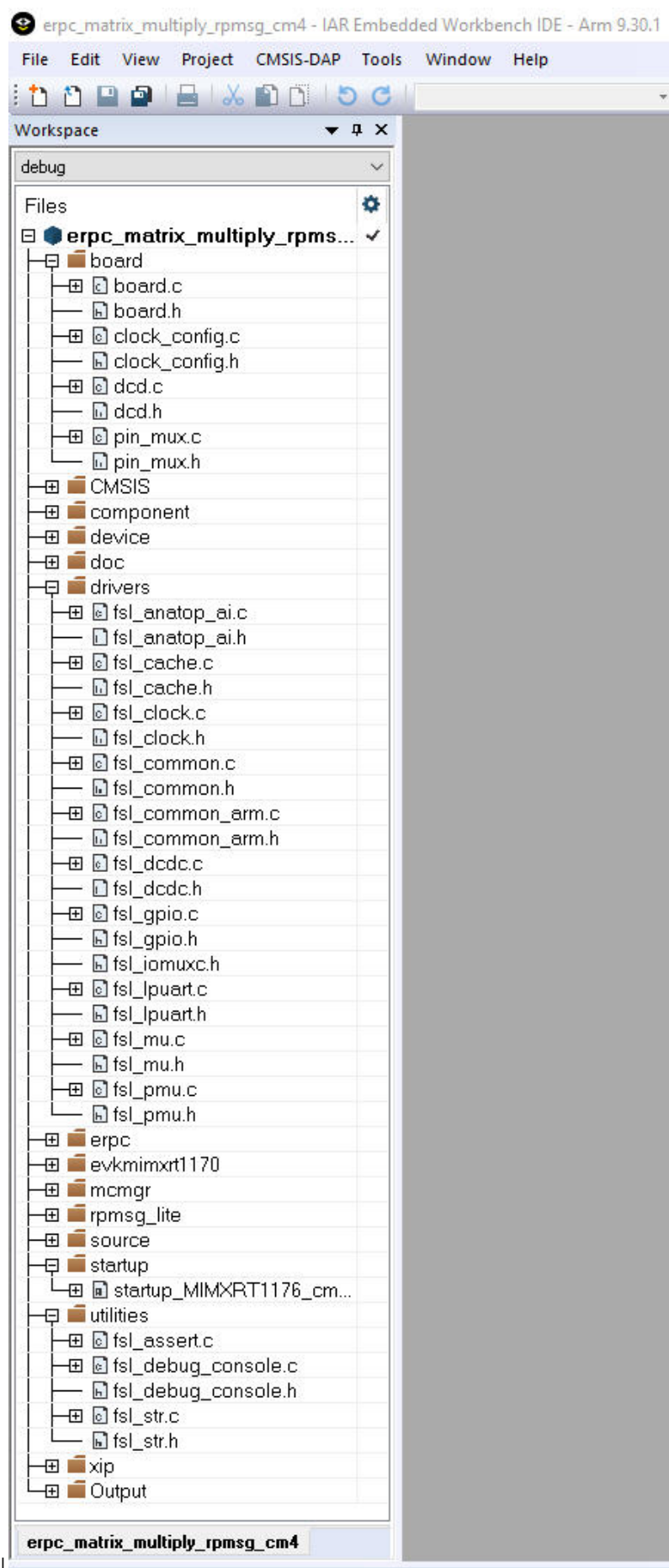
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

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

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



|

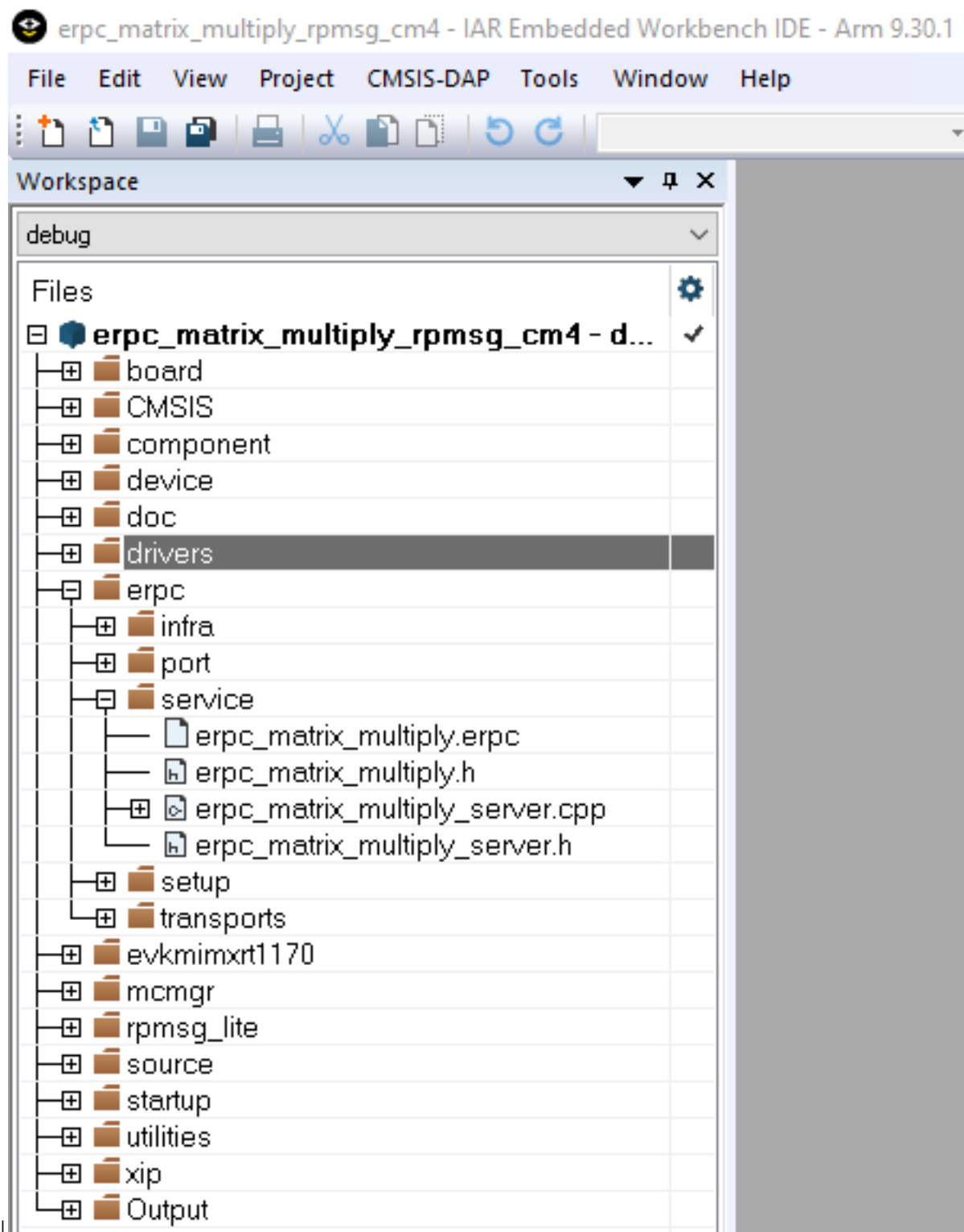
Parent topic:Multicore server application

Server related generated files The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/s



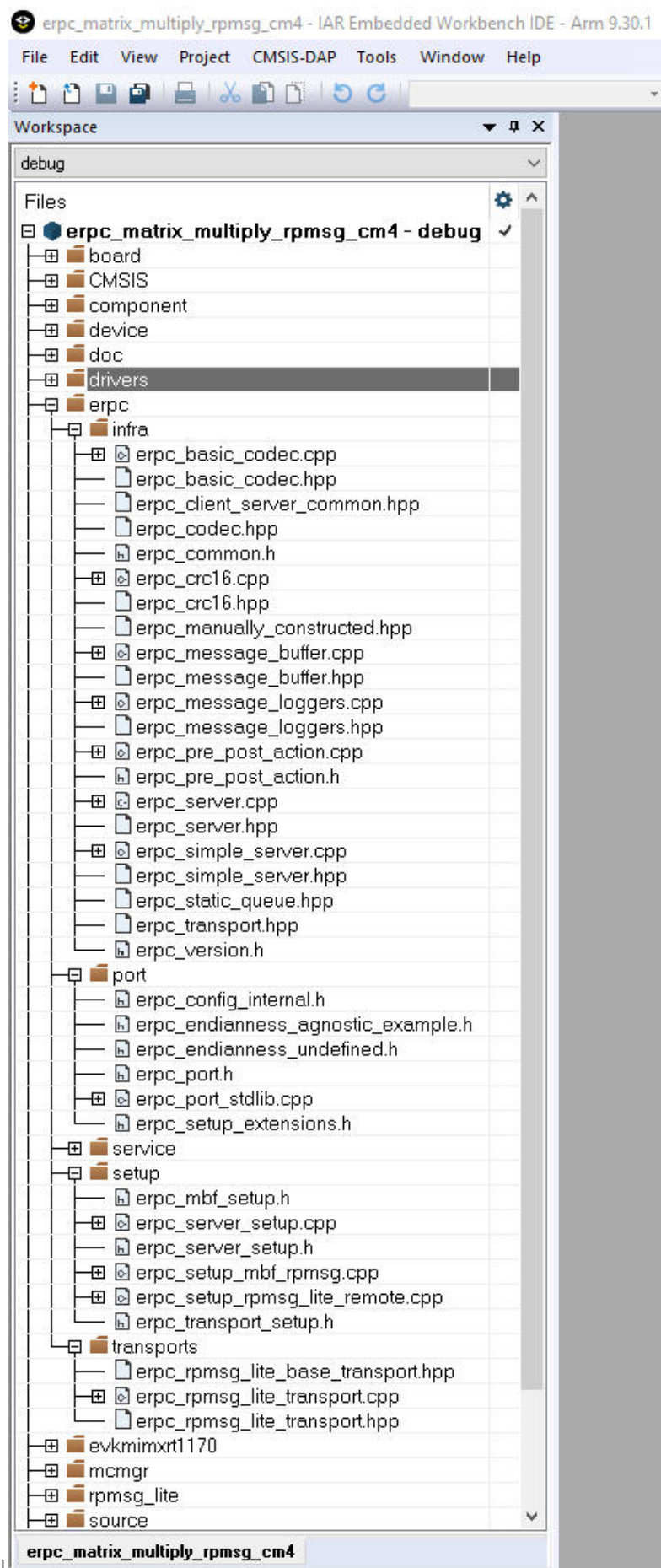
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

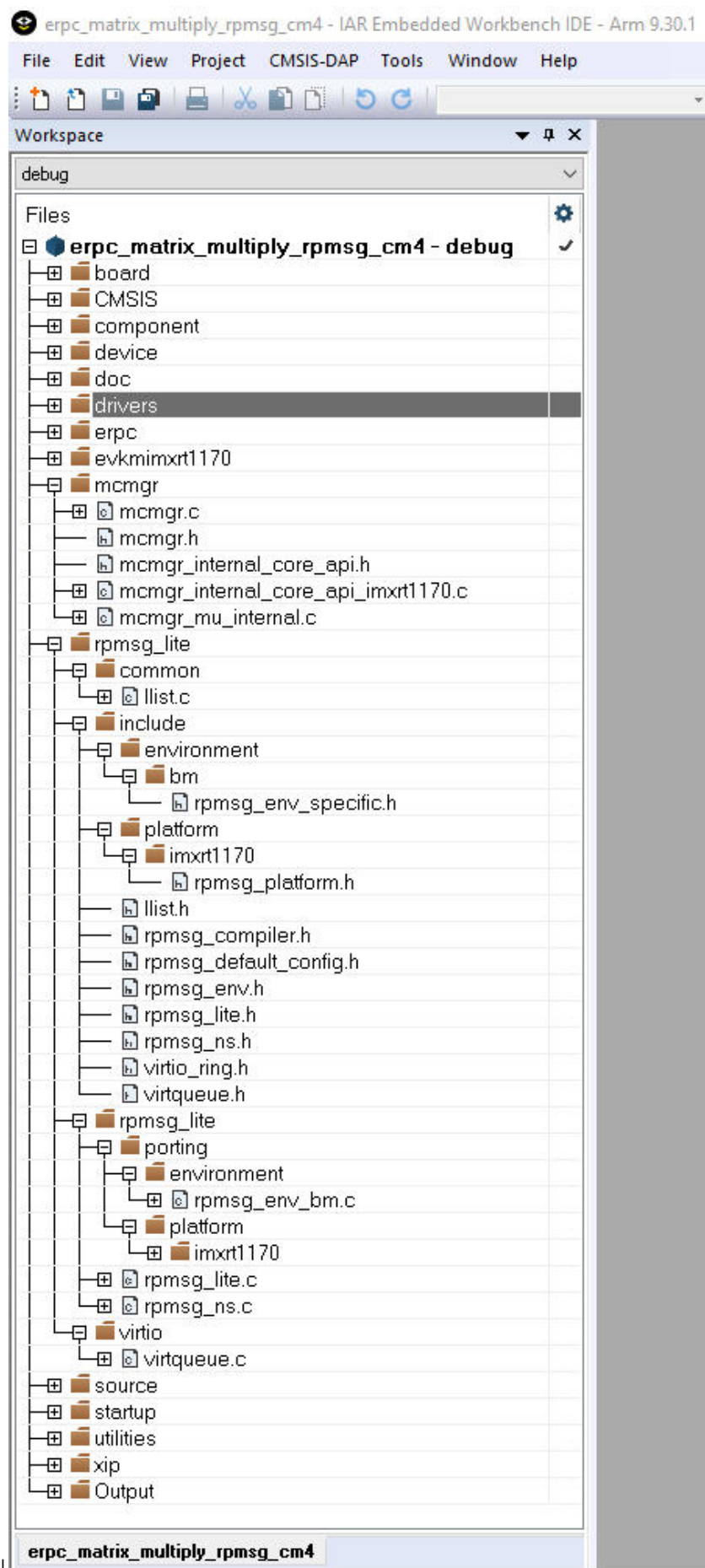
Parent topic:Multicore server application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



Parent topic:Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

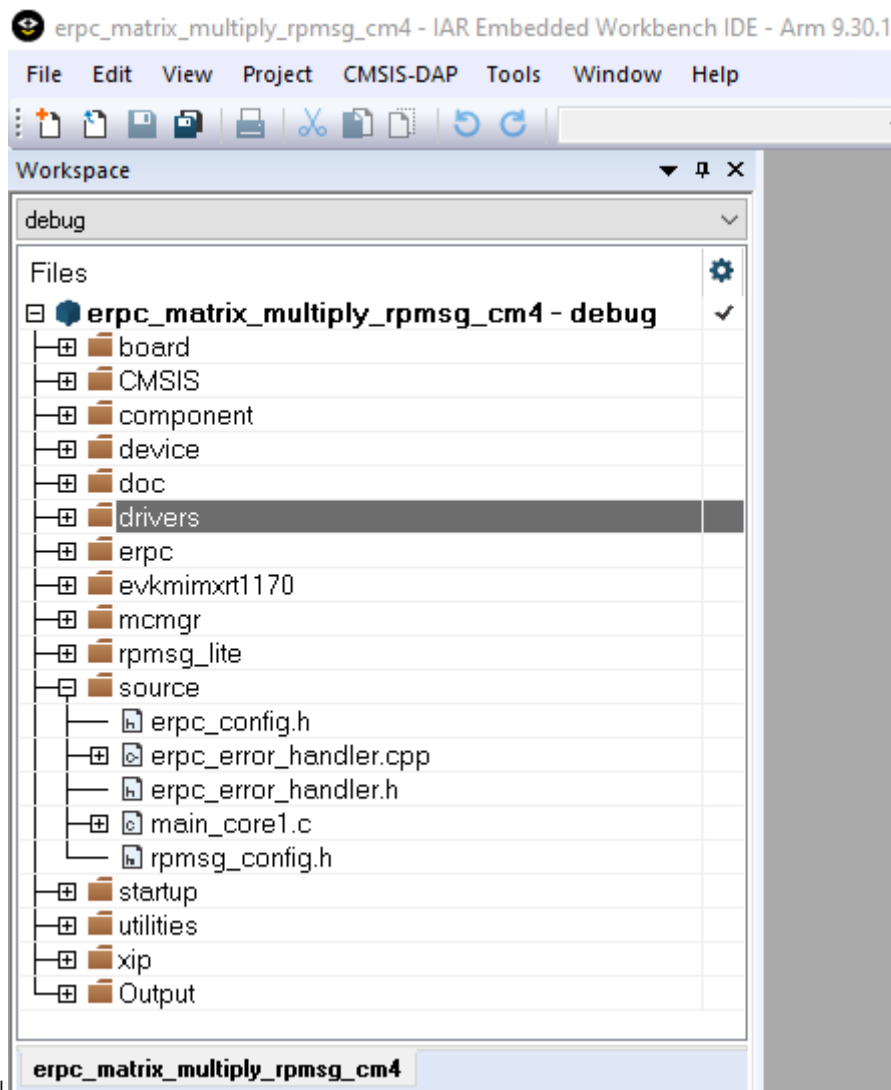
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg` folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

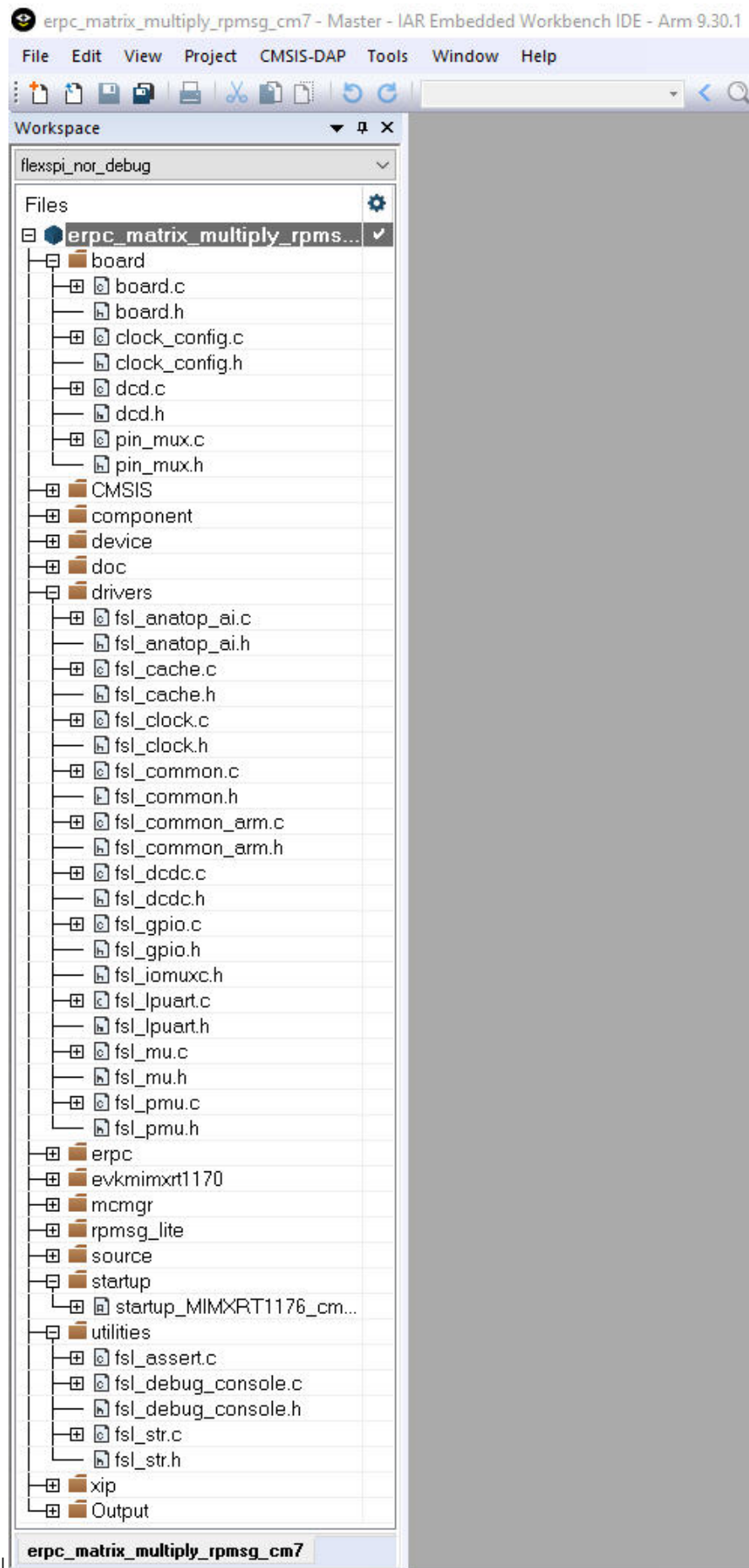
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar`

Project files for the eRPC client have the `_cm7` suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



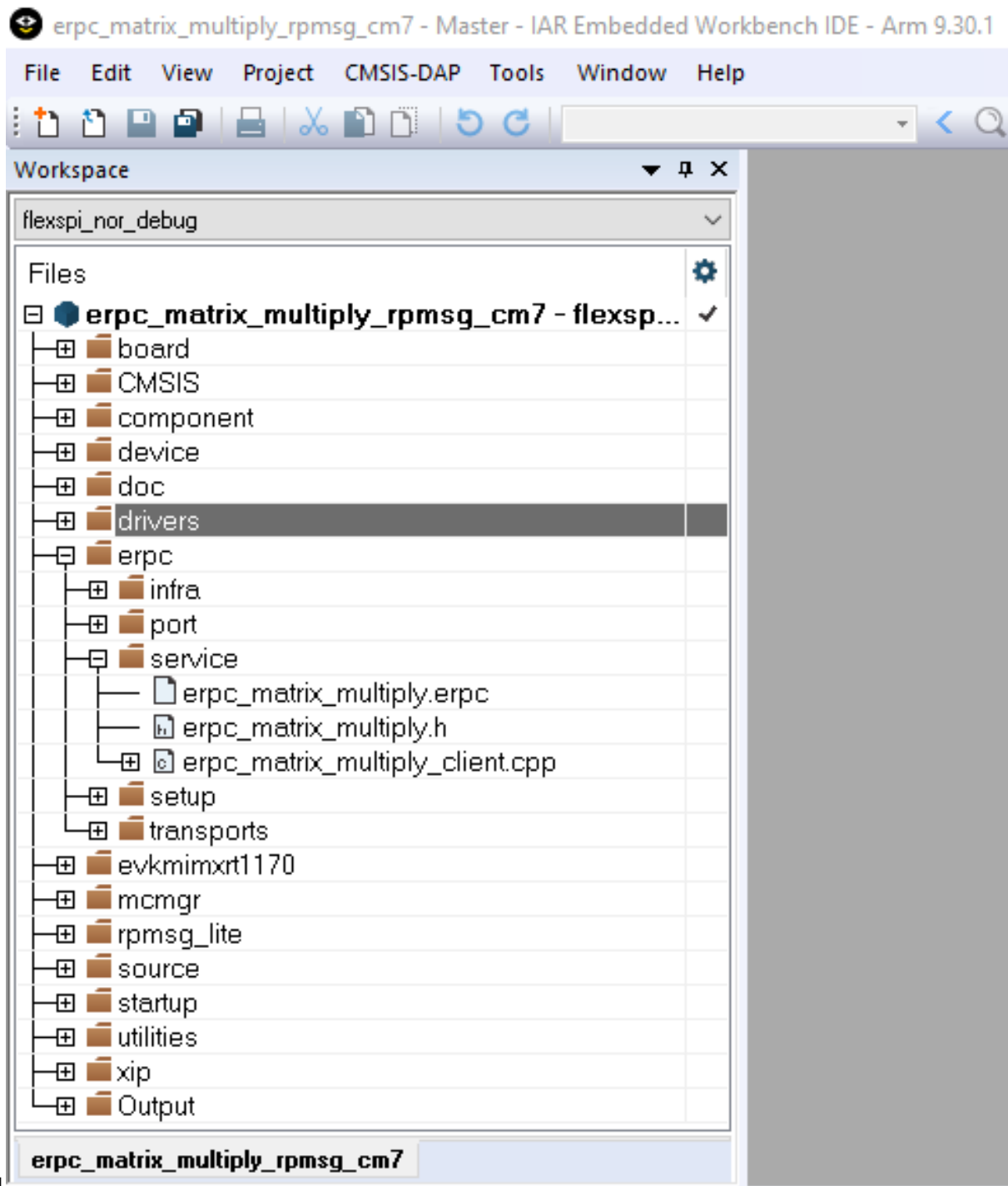
|

Parent topic:Multicore client application

Client-related generated files The client-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/ folder.



Parent topic:Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

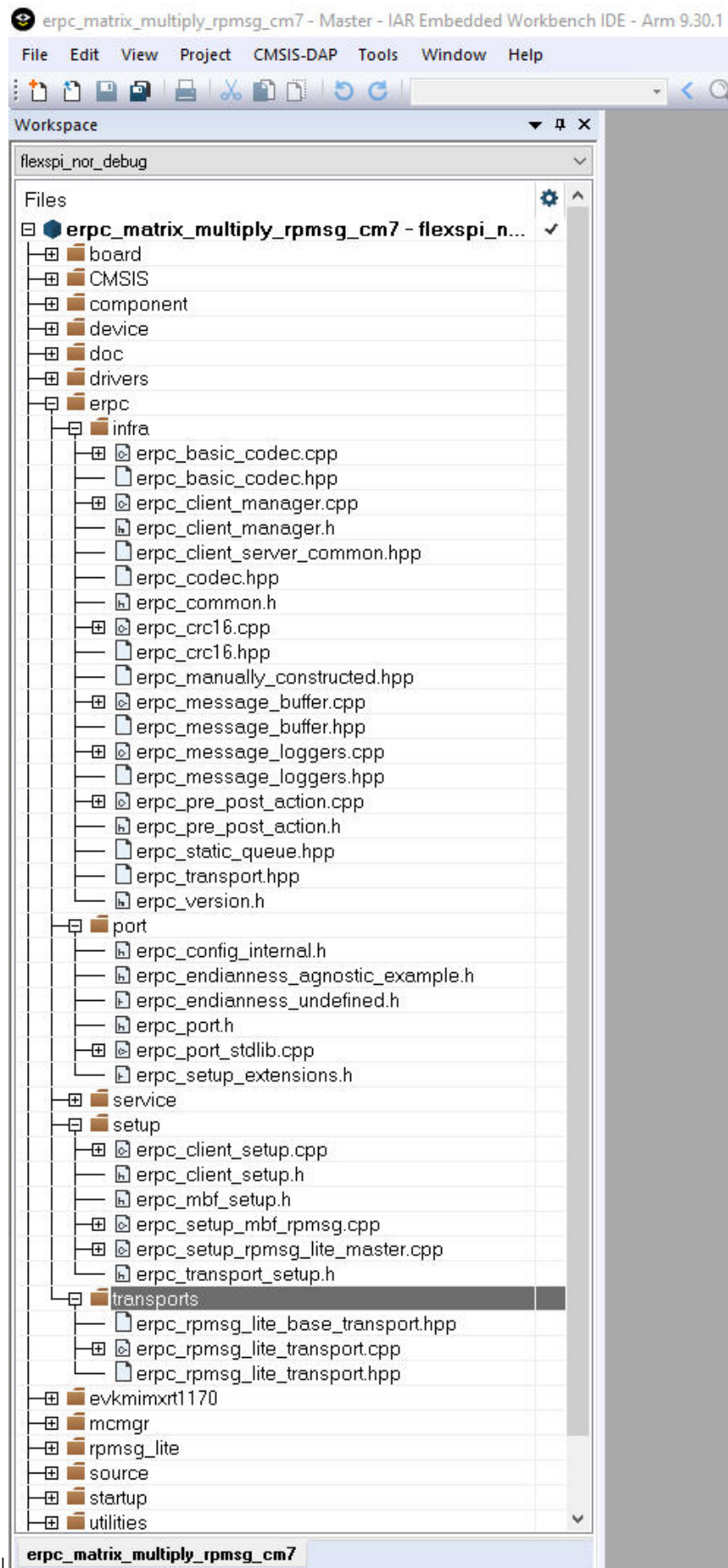
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

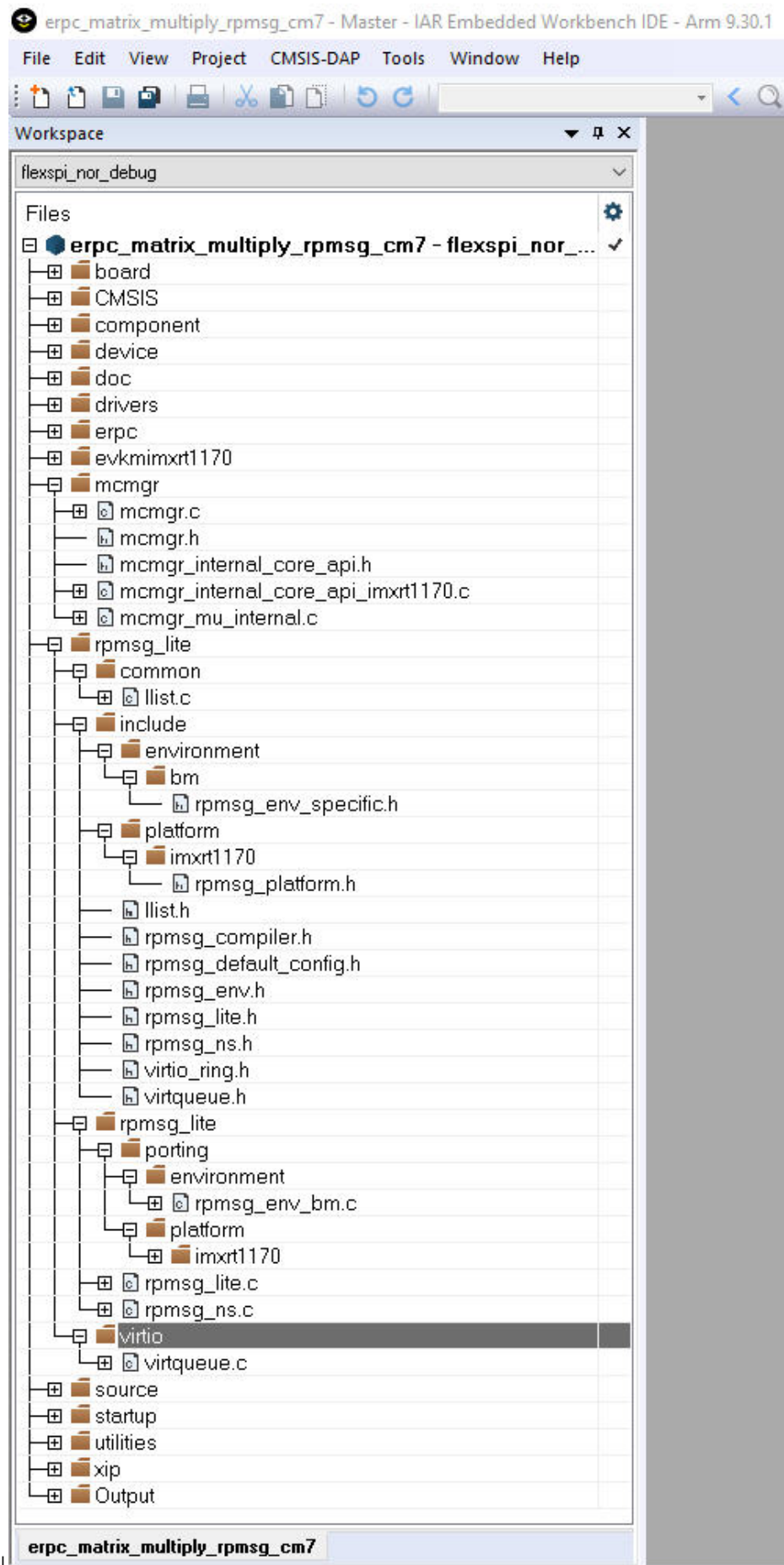
Parent topic:Multicore client application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

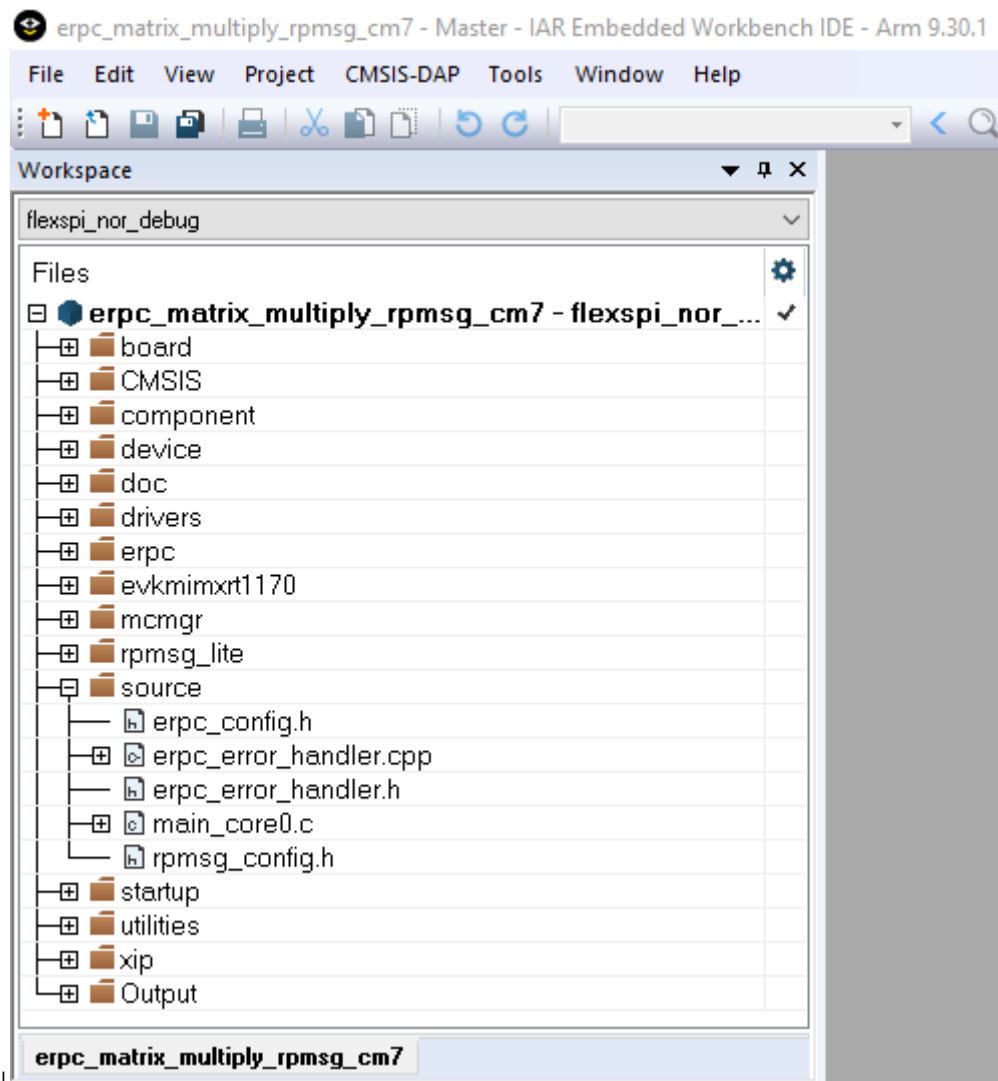
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPSMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPSMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic: Multicore client application

Parent topic: [Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver.
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPSMsg-Lite). The RPSMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
	<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.hpp
	<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
	<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

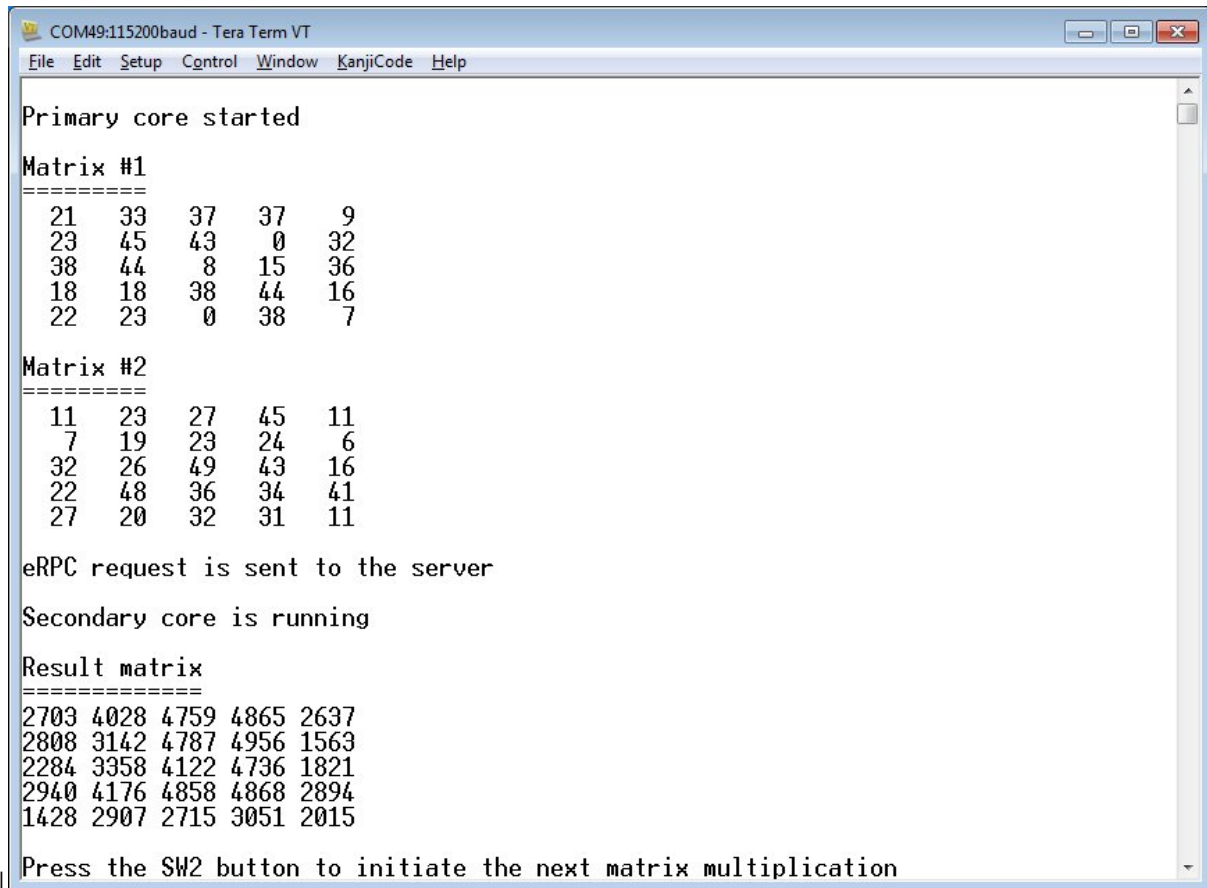
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

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

Primary core started

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

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

eRPC request is sent to the server

Secondary core is running

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

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

eRPC example This section shows how to create an example eRPC application called “Matrix multiply”, which implements one eRPC function (matrix multiply) with two function parameters (two matrices). The client-side application calls this eRPC function, and the server side performs the multiplication of received matrices. The server side then returns the result.

For example, use the NXP MIMXRT1170-EVK board as the target dual-core platform, and the IAR Embedded Workbench for ARM (EWARM) as the target IDE for developing the eRPC example.

- The primary core (CM7) runs the eRPC client.
- The secondary core (CM4) runs the eRPC server.
- RPMsg-Lite (Remote Processor Messaging Lite) is used as the eRPC transport layer.

The “Matrix multiply” application can be also run in the multi-processor setup. In other words, the eRPC client running on one SoC communicates with the eRPC server that runs on another SoC, utilizing different transport channels. It is possible to run the board-to-PC example (PC as the eRPC server and a board as the eRPC client, and vice versa) and also the board-to-board example. These multiprocessor examples are prepared for selected boards only.

| Multicore application source and project files | `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore/`
 | Multiprocessor application source and project files | `<MCUXpressoSDK_install_dir>/boards/<board_name>/multi`
`<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<tr`
 | | eRPC source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/` | | RPLite
 source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/rplite/`

Designing the eRPC application The matrix multiply application is based on calling single eRPC function that takes 2 two-dimensional arrays as input and returns matrix multiplication results as another 2 two-dimensional array. The IDL file syntax supports arrays with the dimension length set by the number only (in the current eRPC implementation). Because of this, a variable is declared in the IDL dedicated to store information about matrix dimension length, and to allow easy maintenance of the user and server code.

For a simple use of the two-dimensional array, the alias name (new type definition) for this data type has been declared in the IDL. Declaring this alias name ensures that the same data type can be used across the client and server applications.

Parent topic: [eRPC example](#)

Creating the IDL file The created IDL file is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/`

The created IDL file contains the following code:

```
program erpc_matrix_multiply
/*! This const defines the matrix size. The value has to be the same as the
Matrix array dimension. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
const int32 matrix_size = 5;
/*! This is the matrix array type. The dimension has to be the same as the
matrix size const. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
type Matrix = int32[matrix_size][matrix_size];
interface MatrixMultiplyService {
erpcMatrixMultiply(in Matrix matrix1, in Matrix matrix2, out Matrix result_matrix) ->
void
}
```

Details:

- The IDL file starts with the program name (*erpc_matrix_multiply*), and this program name is used in the naming of all generated outputs.
- The declaration and definition of the constant variable named *matrix_size* follows next. The *matrix_size* variable is used for passing information about the length of matrix dimensions to the client/server user code.
- The alias name for the two-dimensional array type (*Matrix*) is declared.
- The interface group *MatrixMultiplyService* is located at the end of the IDL file. This interface group contains only one function declaration *erpcMatrixMultiply*.
- As shown above, the function’s declaration contains three parameters of *Matrix* type: *matrix1* and *matrix2* are input parameters, while *result_matrix* is the output parameter. Additionally, the returned data type is declared as *void*.

When writing the IDL file, the following order of items is recommended:

1. Program name at the top of the IDL file.
2. New data types and constants declarations.
3. Declarations of interfaces and functions at the end of the IDL file.

Parent topic: [eRPC example](#)

Using the eRPC generator tool | Windows OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x64`
| Linux OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86`
`<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86`
| | Mac OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Mac`

The files for the “Matrix multiply” example are pre-generated and already a part of the application projects. The following section describes how they have been created.

- The easiest way to create the shim code is to copy the erpcgen application to the same folder where the IDL file (*.erpc) is located; then run the following command:

```
erpcgen <IDL_file>.erpc
```

- In the “Matrix multiply” example, the command should look like:

```
erpcgen erpc_matrix_multiply.erpc
```

Additionally, another method to create the shim code is to execute the eRPC application using input commands:

- “-?”/”—help” – Shows supported commands.
- “-o <filePath>”/”—output<filePath>” – Sets the output directory.

For example,

```
<path_to_erpcgen>/erpcgen -o <path_to_output>  
<path_to_IDL>/<IDL_file_name>.erpc
```

For the “Matrix multiply” example, when the command is executed from the default erpcgen location, it looks like:

```
erpcgen -o
```

```
../../../../../boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service  
../../../../../boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/erpc_matrix_mu
```

In both cases, the following four files are generated into the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service` folder:

- `erpc_matrix_multiply.h`
- `erpc_matrix_multiply_client.cpp`
- `erpc_matrix_multiply_server.h`
- `erpc_matrix_multiply_server.cpp`

For multiprocessor examples, the eRPC file and pre-generated files can be found in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_common/erpc_matrix_multiply/service` folder.

For Linux OS users:

- Do not forget to set the permissions for the eRPC generator application.
- Run the application as `./erpcgen...` instead of as `erpcgen ....`

Parent topic: [eRPC example](#)

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

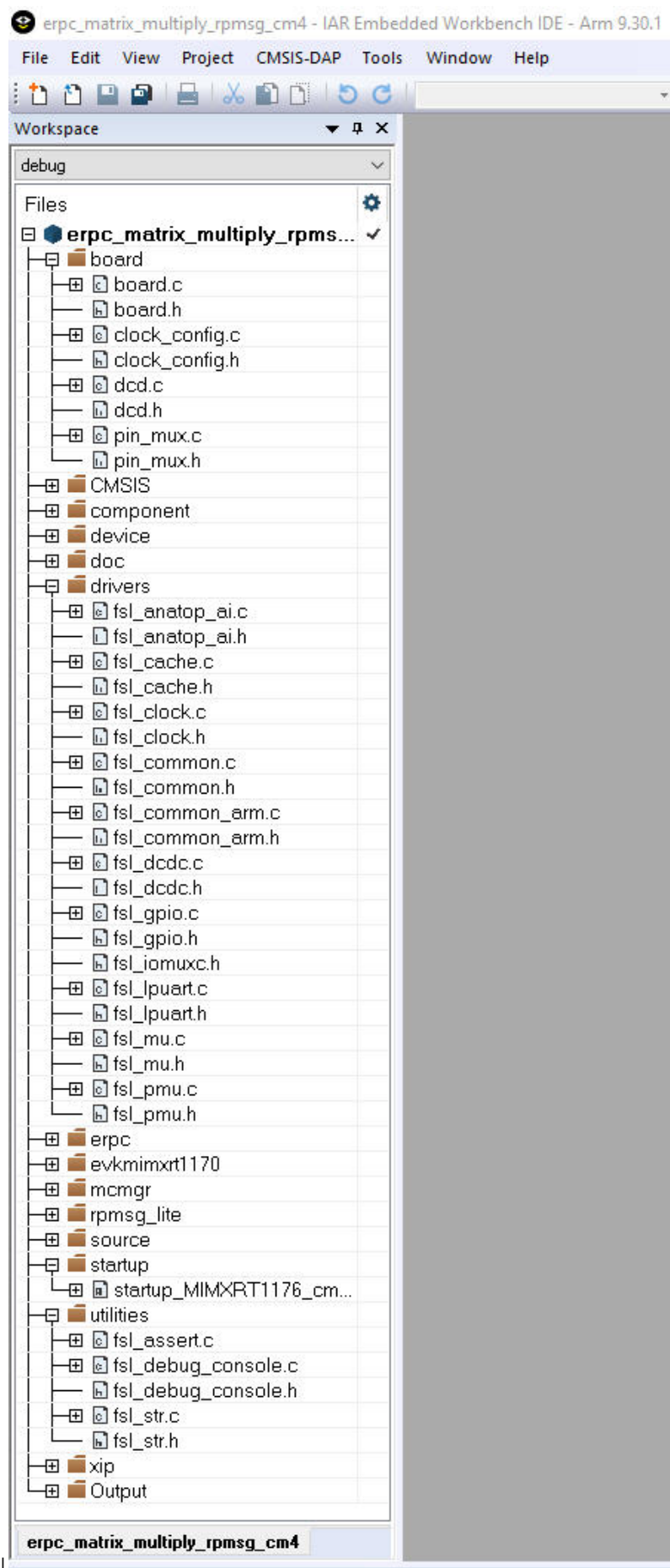
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

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

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



|

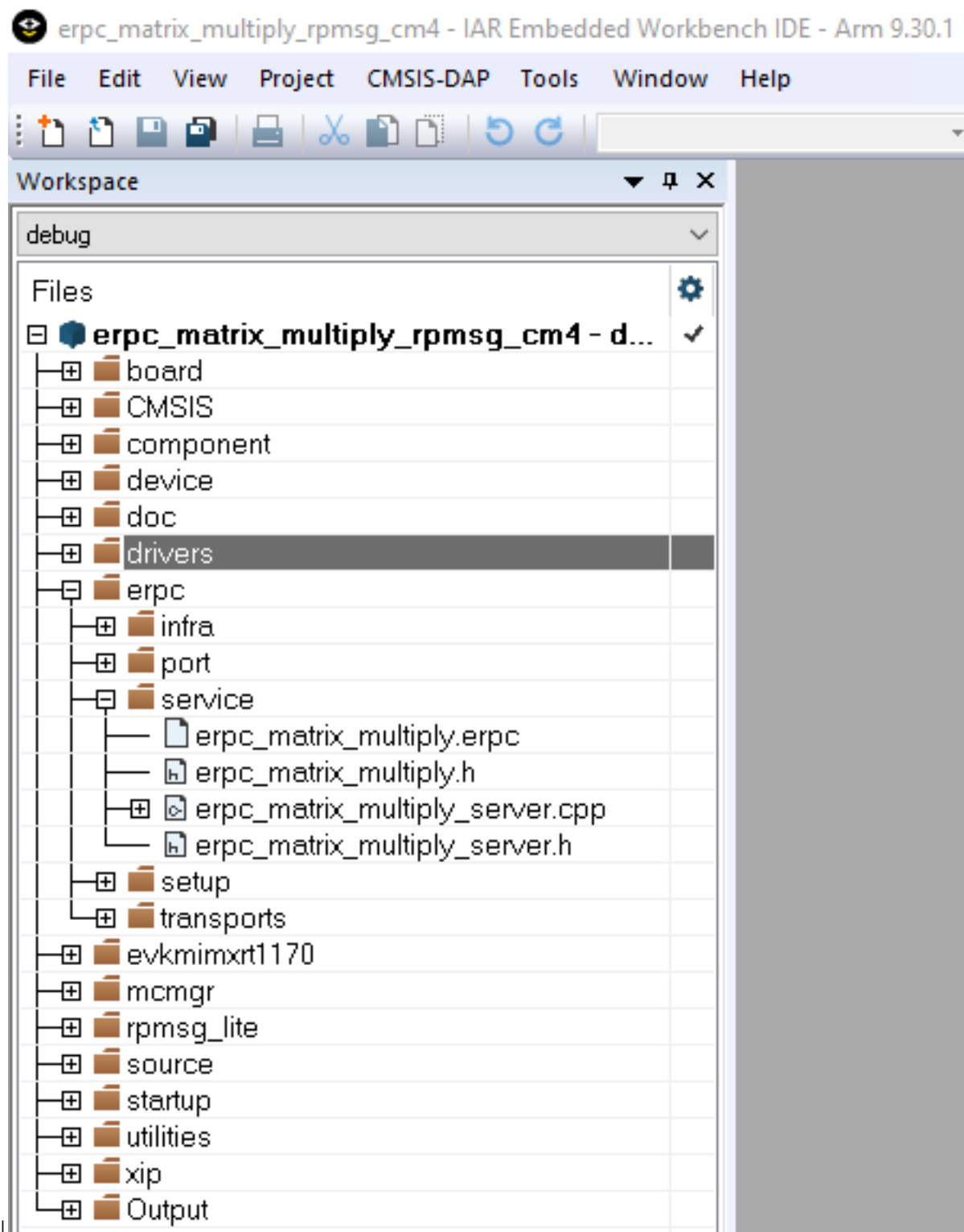
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/



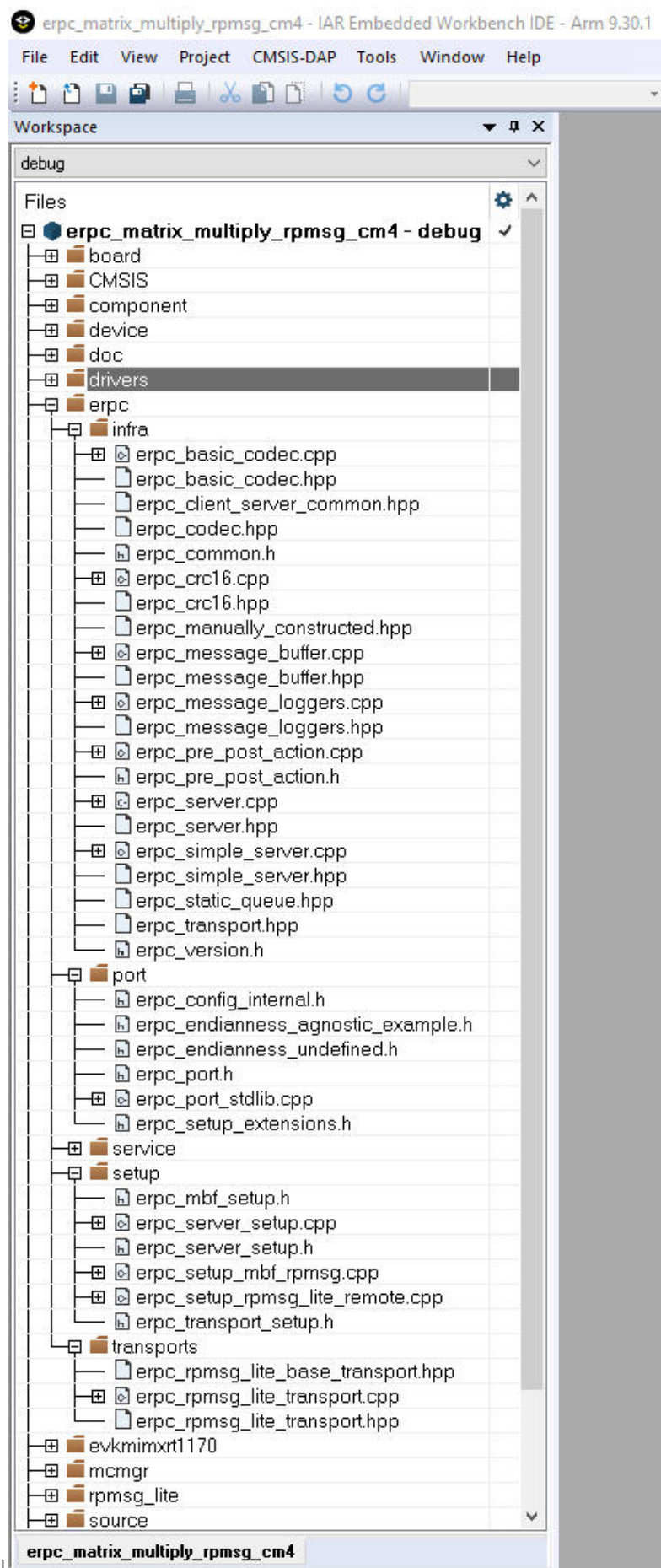
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

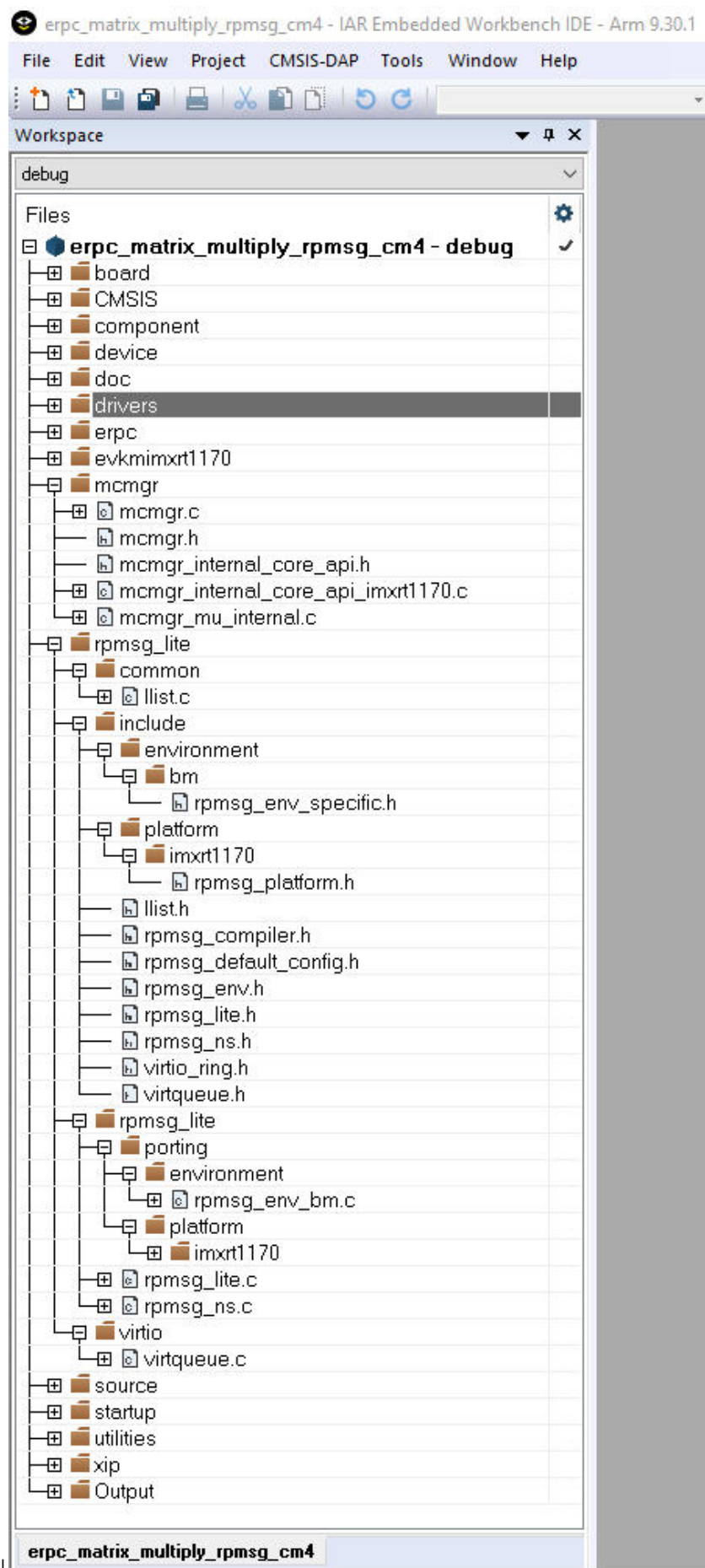
Parent topic: Multicore server application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic:Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

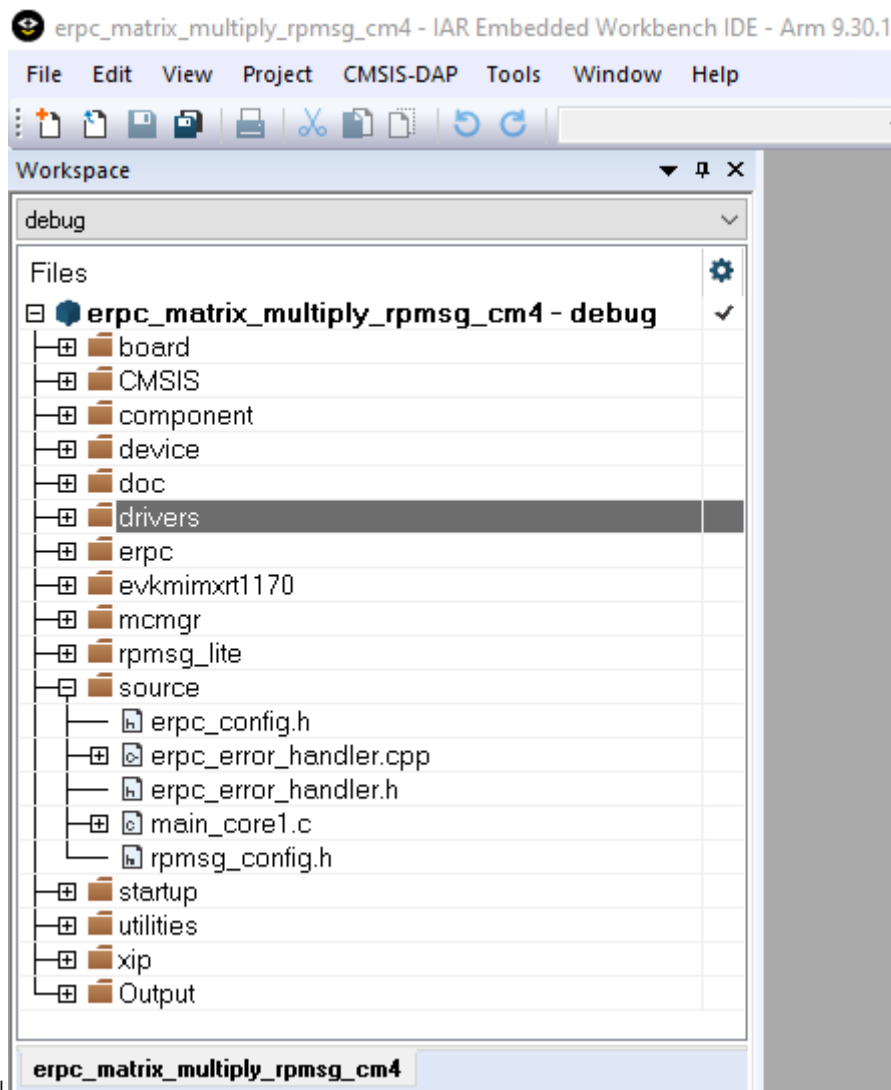
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg` folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

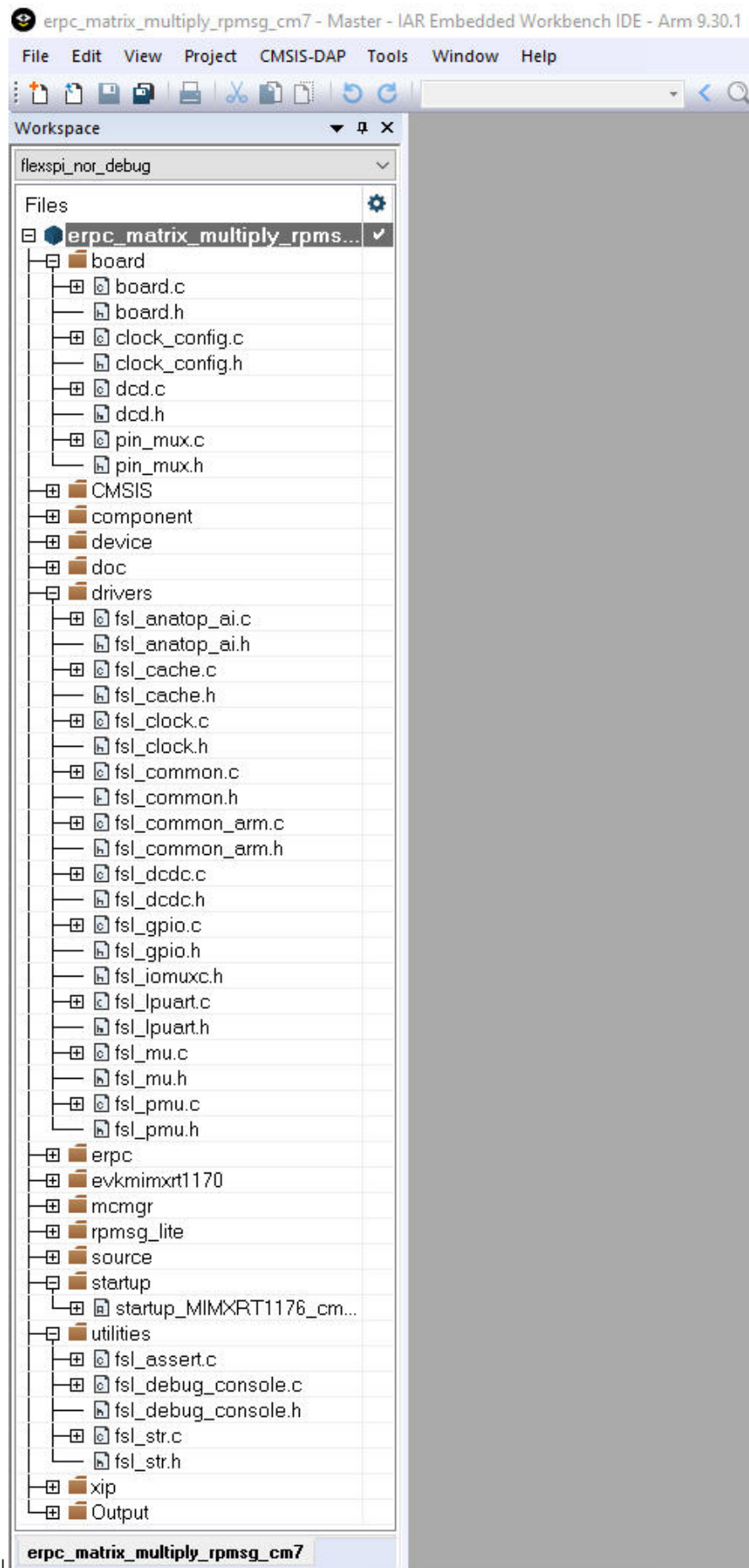
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar`

Project files for the eRPC client have the `_cm7` suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`

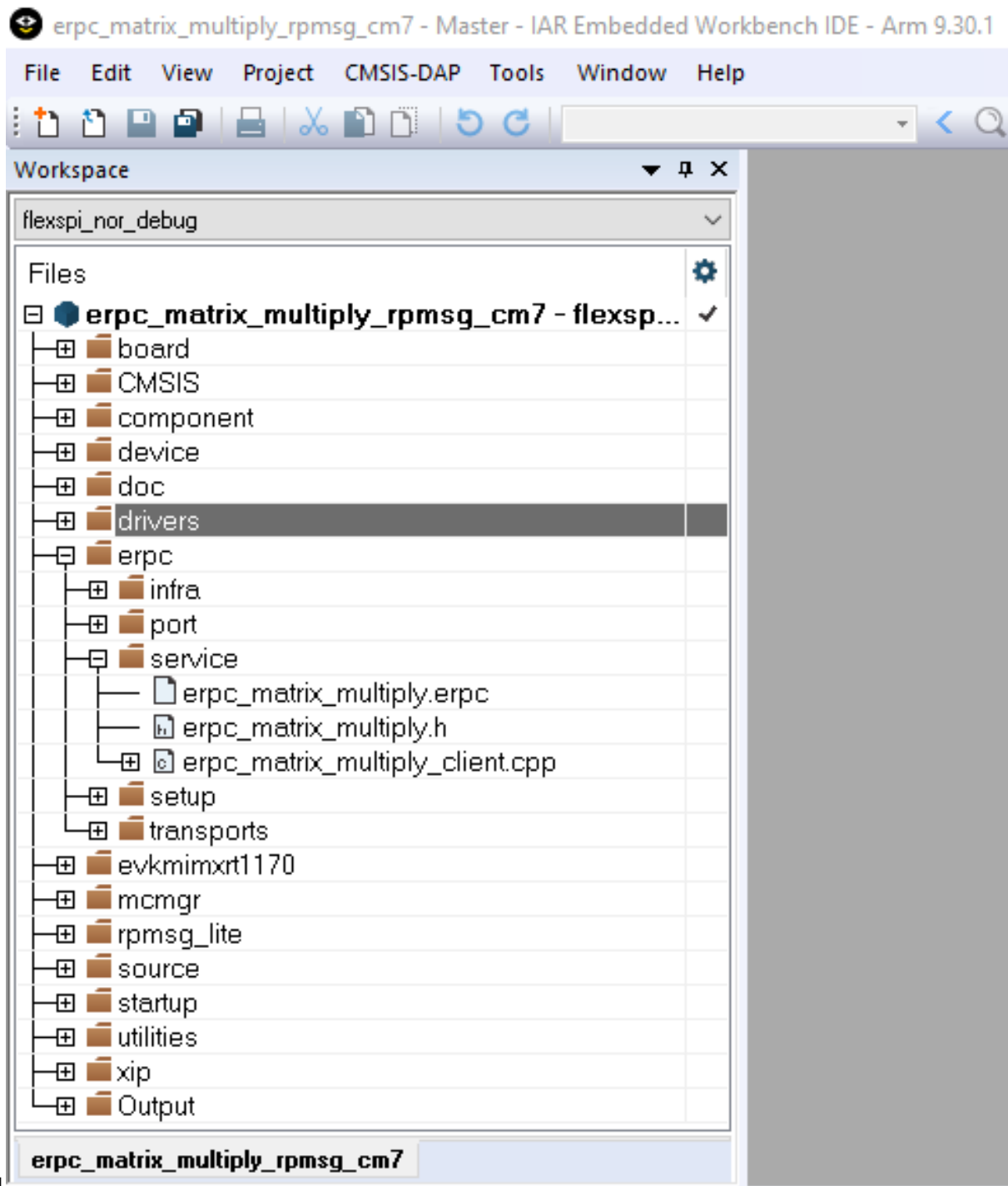


|

Parent topic:Multicore client application**Client-related generated files** The client-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/` folder.



Parent topic:Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

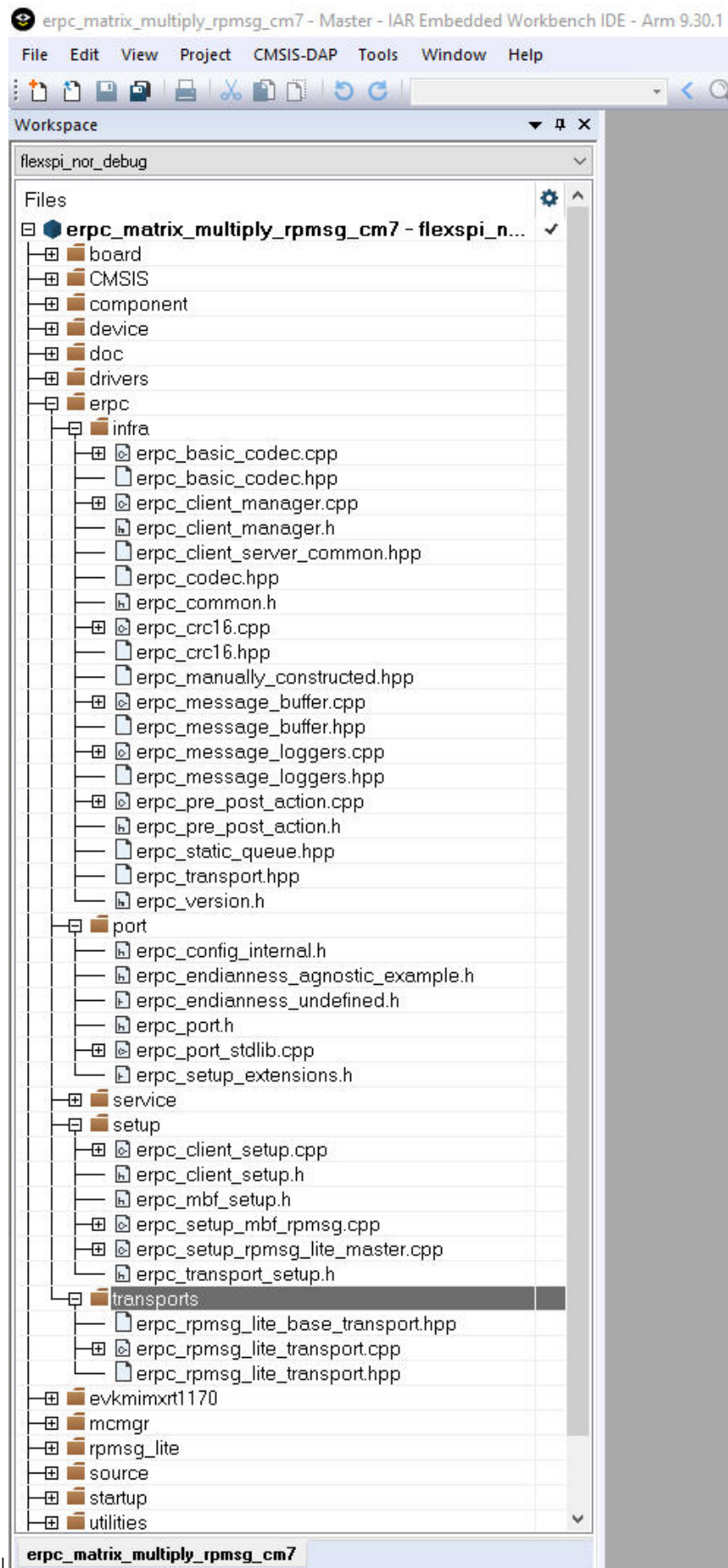
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

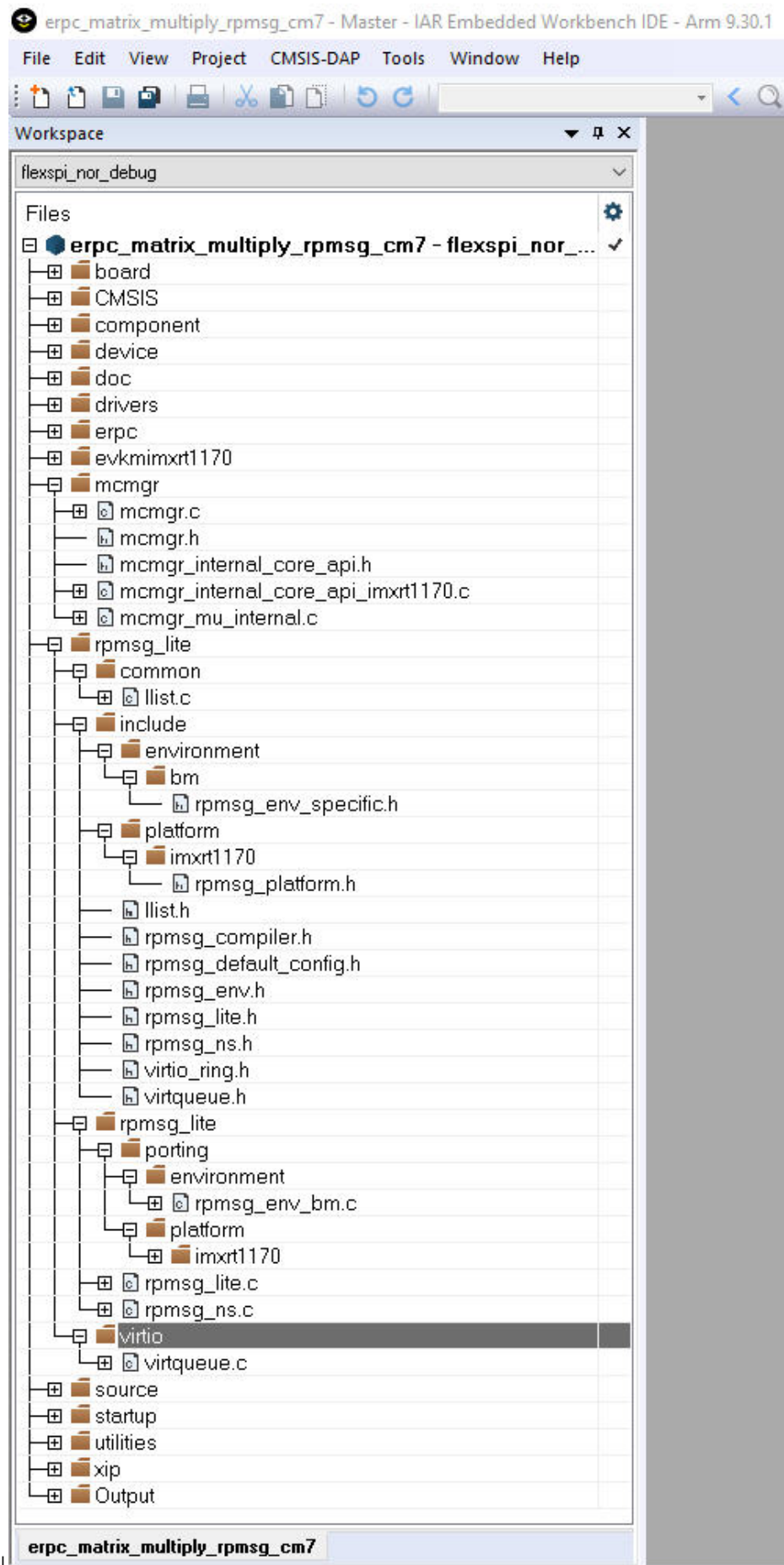
Parent topic: Multicore client application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

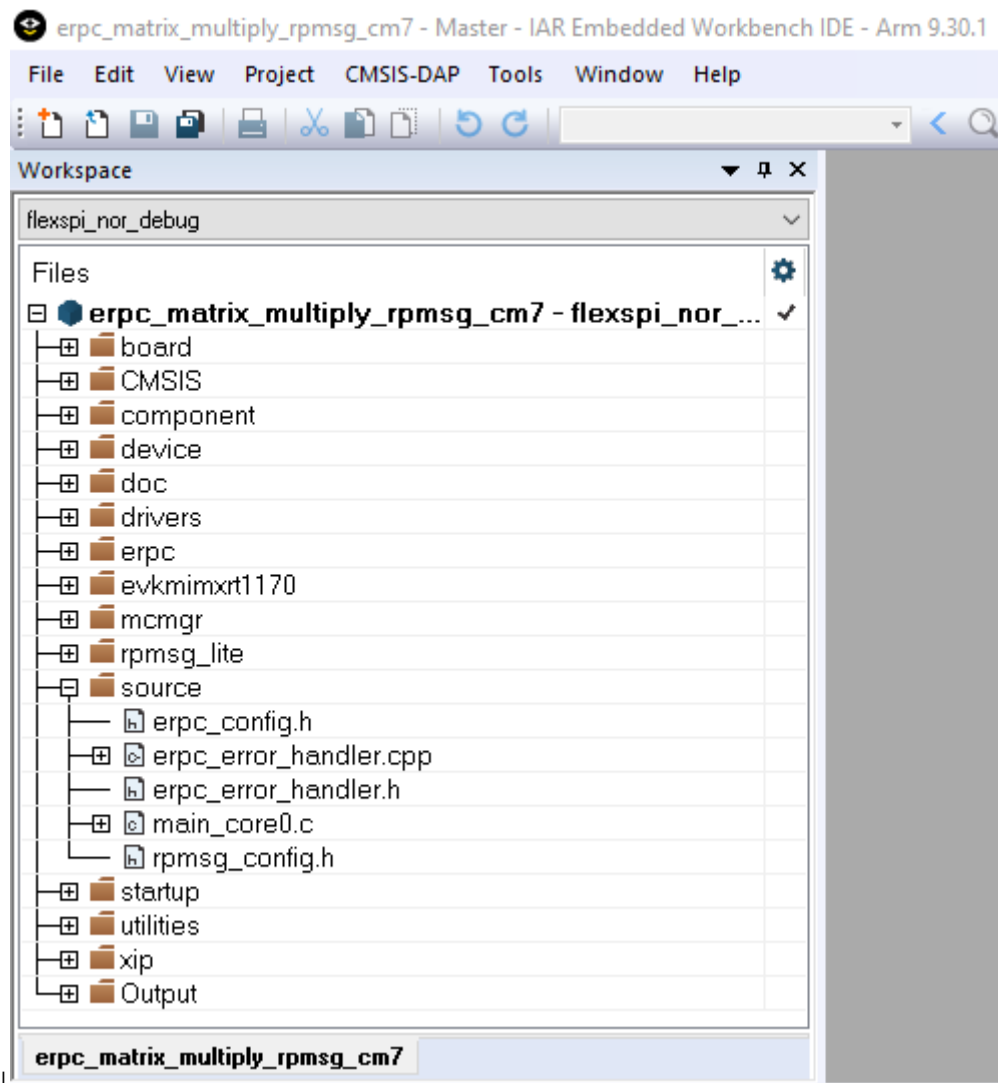
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic: Multicore client application

Parent topic: [Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMs-Lite). The RPMs-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver.
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPSMsg-Lite). The RPSMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
	<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.hpp
	<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
	<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

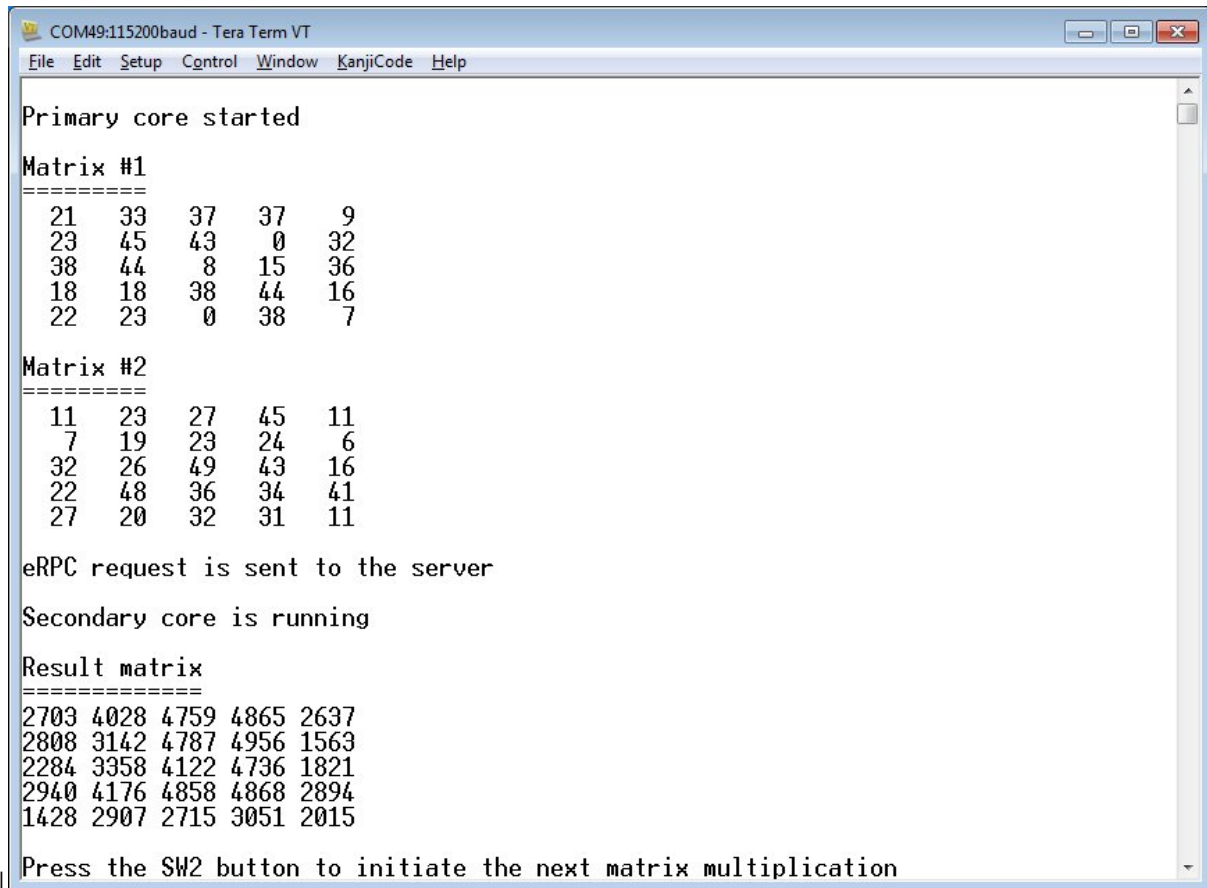
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

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

Primary core started

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

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

eRPC request is sent to the server

Secondary core is running

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

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

Other uses for an eRPC implementation The eRPC implementation is generic, and its use is not limited to just embedded applications. When creating an eRPC application outside the embedded world, the same principles apply. For example, this manual can be used to create an eRPC application for a PC running the Linux operating system. Based on the used type of transport medium, existing transport layers can be used, or new transport layers can be implemented.

For more information and erpc updates see the github.com/EmbeddedRPC.

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Changelog eRPC All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Added

Fixed

- Python code of the eRPC infrastructure was updated to match the proper python code style, add type annotations and improve readability.

1.14.0

Added

- Added Cmake/Kconfig support.
- Made java code jdk11 compliant, GitHub PR #432.
- Added imxrt1186 support into mu transport layer.
- erpcgen: Added assert for listType before usage, GitHub PR #406.

Fixed

- eRPC: Sources reformatted.
- erpc: Fixed typo in semaphore get (mutex -> semaphore), and write it can fail in case of timeout, GitHub PR #446.
- erpc: Free the arbitrated client token from client manager, GitHub PR #444.

- erpc: Fixed Makefile, install the erpc_simple_server header, GitHub PR #447.
- erpc_python: Fixed possible AttributeError and OSError on calling TCPTransport.close(), GitHub PR #438.
- Examples and tests consolidated.

1.13.0

Added

- erpc: Add BSD-3 license to endianness agnostic files, GitHub PR #417.
- eRPC: Add new Zephyr-related transports (zephyr_uart, zephyr_mbox).
- eRPC: Add new Zephyr-related examples.

Fixed

- eRPC,erpcgen: Fixing/improving markdown files, GitHub PR #395.
- eRPC: Fix Python client TCPTransports not being able to close, GitHub PR #390.
- eRPC,erpcgen: Align switch brackets, GitHub PR #396.
- erpc: Fix zephyr uart transport, GitHub PR #410.
- erpc: UART ZEPHYR Transport stop to work after a few transactions when using USB-CDC resolved, GitHub PR #420.

Removed

- eRPC,erpcgen: Remove cstbool library, GitHub PR #403.

1.12.0

Added

- eRPC: Add dynamic/static option for transport init, GitHub PR #361.
- eRPC,erpcgen: Winsock2 support, GitHub PR #365.
- eRPC,erpcgen: Feature/support multiple clients, GitHub PR #271.
- eRPC,erpcgen: Feature/buffer head - Framed transport header data stored in Message-Buffer, GitHub PR #378.
- eRPC,erpcgen: Add experimental Java support.

Fixed

- eRPC: Fix receive error value for spidev, GitHub PR #363.
- eRPC: UartTransport::init adaptation to changed driver.
- eRPC: Fix typo in assert, GitHub PR #371.
- eRPC,erpcgen: Move enums to enum classes, GitHub PR #379.
- eRPC: Fixed rpmsg tty transport to work with serial transport, GitHub PR #373.

1.11.0

Fixed

- eRPC: Makefiles update, GitHub PR #301.
- eRPC: Resolving warnings in Python, GitHub PR #325.
- eRPC: Python3.8 is not ready for usage of typing.Any type, GitHub PR #325.
- eRPC: Improved codec function to use reference instead of address, GitHub PR #324.
- eRPC: Fix NULL check for pending client creation, GitHub PR #341.
- eRPC: Replace sprintf with snprintf, GitHub PR #343.
- eRPC: Use MU_SendMsg blocking call in MU transport.
- eRPC: New LPSPI and LPI2C transport layers.
- eRPC: Freeing static objects, GitHub PR #353.
- eRPC: Fixed casting in deinit functions, GitHub PR #354.
- eRPC: Align LIBUSB_SIO.GetNumPorts API use with libusb_sio python module v. 2.1.11.
- erpcgen: Renamed temp variable to more generic one, GitHub PR #321.
- erpcgen: Add check that string read is not more than max length, GitHub PR #328.
- erpcgen: Move to g++ in pytest, GitHub PR #335.
- erpcgen: Use build=release for make, GitHub PR #334.
- erpcgen: Removed boost dependency, GitHub PR #346.
- erpcgen: Mingw support, GitHub PR #344.
- erpcgen: VS build update, GitHub PR #347.
- erpcgen: Modified name for common types macro scope, GitHub PR #337.
- erpcgen: Fixed memcpy for template, GitHub PR #352.
- eRPC,erpcgen: Change default build target to release + adding artefacts, GitHub PR #334.
- eRPC,erpcgen: Remove redundant includes, GitHub PR #338.
- eRPC,erpcgen: Many minor code improvements, GitHub PR #323.

1.10.0

Fixed

- eRPC: MU transport layer switched to blocking MU_SendMsg() API use.

1.10.0

Added

- eRPC: Add TCP_NODELAY option to python, GitHub PR #298.

Fixed

- eRPC: MUPort adaptation to new supported SoCs.
- eRPC: Simplifying CI with installing dependencies using shell script, GitHub PR #267.
- eRPC: Using event for waiting for sock connection in TCP python server, formatting python code, C specific includes, GitHub PR #269.
- eRPC: Endianness agnostic update, GitHub PR #276.
- eRPC: Assertion added for functions which are returning status on freeing memory, GitHub PR #277.
- eRPC: Fixed closing arbitrator server in unit tests, GitHub PR #293.
- eRPC: Makefile updated to reflect the correct header names, GitHub PR #295.
- eRPC: Compare value length to used length() in reading data from message buffer, GitHub PR #297.
- eRPC: Replace EXPECT_TRUE with EXPECT_EQ in unit tests, GitHub PR #318.
- eRPC: Adapt rpmsg_lite based transports to changed rpmsg_lite_wait_for_link_up() API parameters.
- eRPC, erpcgen: Better distinguish which file can and cannot be linked by C linker, GitHub PR #266.
- eRPC, erpcgen: Stop checking if pointer is NULL before sending it to the erpc_free function, GitHub PR #275.
- eRPC, erpcgen: Changed api to count with more interfaces, GitHub PR #304.
- erpcgen: Check before reading from heap the buffer boundaries, GitHub PR #287.
- erpcgen: Several fixes for tests and CI, GitHub PR #289.
- erpcgen: Refactoring erpcgen code, GitHub PR #302.
- erpcgen: Fixed assigning const value to enum, GitHub PR #309.
- erpcgen: Enable runTesttest_enumErrorCode_allDirection, serialize enums as int32 instead of uint32.

1.9.1

Fixed

- eRPC: Construct the USB CDC transport, rather than a client, GitHub PR #220.
- eRPC: Fix premature import of package, causing failure when attempting installation of Python library in a clean environment, GitHub PR #38, #226.
- eRPC: Improve python detection in make, GitHub PR #225.
- eRPC: Fix several warnings with deprecated call in pytest, GitHub PR #227.
- eRPC: Fix freeing union members when only default need be freed, GitHub PR #228.
- eRPC: Fix making test under Linux, GitHub PR #229.
- eRPC: Assert costumizing, GitHub PR #148.
- eRPC: Fix corrupt clientList bug in TransportArbitrator, GitHub PR #199.
- eRPC: Fix build issue when invoking g++ with -Wno-error=free-nonheap-object, GitHub PR #233.
- eRPC: Fix inout cases, GitHub PR #237.

- eRPC: Remove ERPC_PRE_POST_ACTION dependency on return type, GitHub PR #238.
- eRPC: Adding NULL to ptr when codec function failed, fixing memcpy when fail is present during deserialization, GitHub PR #253.
- eRPC: MessageBuffer usage improvement, GitHub PR #258.
- eRPC: Get rid of serial and enum34 dependency (enum34 is in python3 since 3.4 (from 2014)), GitHub PR #247.
- eRPC: Several MISRA violations addressed.
- eRPC: Fix timeout for Freertos semaphore, GitHub PR #251.
- eRPC: Use of rpmsg_lite_wait_for_link_up() in rpmsg_lite based transports, GitHub PR #223.
- eRPC: Fix codec nullptr dereferencing, GitHub PR #264.
- erpcgen: Fix two syntax errors in erpcgen Python output related to non-encapsulated unions, improved test for union, GitHub PR #206, #224.
- erpcgen: Fix serialization of list/binary types, GitHub PR #240.
- erpcgen: Fix empty list parsing, GitHub PR #72.
- erpcgen: Fix templates for malloc errors, GitHub PR #110.
- erpcgen: Get rid of encapsulated union declarations in global scale, improve enum usage in unions, GitHub PR #249, #250.
- erpcgen: Fix compile error:UniqueIdChecker.cpp:156:104:'sort' was not declared, GitHub PR #265.

1.9.0

Added

- eRPC: Allow used LIBUSBSIO device index being specified from the Python command line argument.

Fixed

- eRPC: Improving template usage, GitHub PR #153.
- eRPC: run_clang_format.py cleanup, GitHub PR #177.
- eRPC: Build TCP transport setup code into liberpc, GitHub PR #179.
- eRPC: Fix multiple definitions of g_client error, GitHub PR #180.
- eRPC: Fix memset past end of buffer in erpc_setup_mbf_static.cpp, GitHub PR #184.
- eRPC: Fix deprecated error with newer pytest version, GitHub PR #203.
- eRPC, erpcgen: Static allocation support and usage of rpmsg static FreeRTOSs related API, GitHub PR #168, #169.
- erpcgen: Remove redundant module imports in erpcgen, GitHub PR #196.

1.8.1

Added

- eRPC: New i2c_slave_transport transport introduced.

Fixed

- eRPC: Fix misra erpc c, GitHub PR #158.
- eRPC: Allow conditional compilation of message_loggers and pre_post_action.
- eRPC: (D)SPI slave transports updated to avoid busy loops in rtos environments.
- erpcgen: Re-implement EnumMember::hasValue(), GitHub PR #159.
- erpcgen: Fixing several misra issues in shim code, erpcgen and unit tests updated, GitHub PR #156.
- erpcgen: Fix bison file, GitHub PR #156.

1.8.0

Added

- eRPC: Support win32 thread, GitHub PR #108.
- eRPC: Add mbed support for malloc() and free(), GitHub PR #92.
- eRPC: Introduced pre and post callbacks for eRPC call, GitHub PR #131.
- eRPC: Introduced new USB CDC transport.
- eRPC: Introduced new Linux spidev-based transport.
- eRPC: Added formatting extension for VSC, GitHub PR #134.
- erpcgen: Introduce ustring type for unsigned char and force cast to char*, GitHub PR #125.

Fixed

- eRPC: Update makefile.
- eRPC: Fixed warnings and error with using MessageLoggers, GitHub PR #127.
- eRPC: Extend error msg for python server service handle function, GitHub PR #132.
- eRPC: Update CMSIS UART transport layer to avoid busy loops in rtos environments, introduce semaphores.
- eRPC: SPI transport update to allow usage without handshaking GPIO.
- eRPC: Native _WIN32 erpc serial transport and threading.
- eRPC: Arbitrator deadlock fix, TCP transport updated, TCP setup functions introduced, GitHub PR #121.
- eRPC: Update of matrix_multiply.py example: Add -serial and -baud argument, GitHub PR #137.
- eRPC: Update of .clang-format, GitHub PR #140.
- eRPC: Update of erpc_framed_transport.cpp: return error if received message has zero length, GitHub PR #141.
- eRPC, erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136, #139.
- eRPC, erpcgen: Core re-formatted using Clang version 10.
- erpcgen: Enable deallocation in server shim code when callback/function pointer used as out parameter in IDL.
- erpcgen: Removed '\$' character from generated symbol name in '_\$union' suffix, GitHub PR #103.

- erpcgen: Resolved mismatch between C++ and Python for callback index type, GitHub PR #111.
- erpcgen: Python generator improvements, GitHub PR #100, #118.
- erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136.

1.7.4

Added

- eRPC: Support MU transport unit testing.
- eRPC: Adding mbed os support.

Fixed

- eRPC: Unit test code updated to handle service add and remove operations.
- eRPC: Several MISRA issues in rpmsg-based transports addressed.
- eRPC: Fixed Linux/TCP acceptance tests in release target.
- eRPC: Minor documentation updates, code formatting.
- erpcgen: Whitespace removed from C common header template.

1.7.3

Fixed

- eRPC: Improved the test_callbacks logic to be more understandable and to allow requested callback execution on the server side.
- eRPC: TransportArbitrator::prepareClientReceive modified to avoid incorrect return value type.
- eRPC: The ClientManager and the ArbitratedClientManager updated to avoid performing client requests when the previous serialization phase fails.
- erpcgen: Generate the shim code for destroy of statically allocated services.

1.7.2

Added

- eRPC: Add missing doxygen comments for transports.

Fixed

- eRPC: Improved support of const types.
- eRPC: Fixed Mac build.
- eRPC: Fixed serializing python list.
- eRPC: Documentation update.

1.7.1

Fixed

- eRPC: Fixed semaphore in static message buffer factory.
- erpcgen: Fixed MU received error flag.
- erpcgen: Fixed tcp transport.

1.7.0

Added

- eRPC: List names are based on their types. Names are more deterministic.
- eRPC: Service objects are as a default created as global static objects.
- eRPC: Added missing doxygen comments.
- eRPC: Added support for 64bit numbers.
- eRPC: Added support of program language specific annotations.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Generating crc value is optional.
- eRPC: Fixed CMSIS Uart driver. Removed dependency on KSDK.
- eRPC: Forbid users use reserved words.
- eRPC: Removed outByref for function parameters.
- eRPC: Optimized code style of callback functions.

1.6.0

Added

- eRPC: Added @nullable support for scalar types.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Improved eRPC nested calls.
- eRPC: Improved eRPC list length variable serialization.

1.5.0

Added

- eRPC: Added support for unions type non-wrapped by structure.
- eRPC: Added callbacks support.
- eRPC: Added support @external annotation for functions.
- eRPC: Added support @name annotation.
- eRPC: Added Messaging Unit transport layer.
- eRPC: Added RPMSG Lite RTOS TTY transport layer.
- eRPC: Added version verification and IDL version verification between eRPC code and eRPC generated shim code.
- eRPC: Added support of shared memory pointer.
- eRPC: Added annotation to forbid generating const keyword for function parameters.
- eRPC: Added python matrix multiply example.
- eRPC: Added nested call support.
- eRPC: Added struct member “byref” option support.
- eRPC: Added support of forward declarations of structures
- eRPC: Added Python RPMsg Multiendpoint kernel module support
- eRPC: Added eRPC sniffer tool

1.4.0**Added**

- eRPC: New RPMsg-Lite Zero Copy (RPMsgZC) transport layer.

Fixed

- eRPC: win_flex_bison.zip for windows updated.
- eRPC: Use one codec (instead of inCodec outCodec).

[1.3.0]**Added**

- eRPC: New annotation types introduced (@length, @max_length, ...).
- eRPC: Support for running both erpc client and erpc server on one side.
- eRPC: New transport layers for (LP)UART, (D)SPI.
- eRPC: Error handling support.

[1.2.0]**Added**

- eRPC source directory organization changed.
- Many eRPC improvements.

[1.1.0]

Added

- Multicore SDK 1.1.0 ported to KSDK 2.0.0.

[1.0.0]

Added

- Initial Release

3.3 Wireless

3.3.1 NXP Wireless Framework and Stacks

Wireless Framework

Wireless Connectivity Framework Connectivity Framework repository provides both connectivity platform enablement with hardware abstraction layer and a set of Services for NXP connectivity stacks : BLE, Zigbee, OpenThread, Matter.

The connectivity framework repository consists of:

- Common folder to common header files for minimal type definition to be used in the repo
- Platform folder used for platform enablement with Hardware abstraction:
 - platform/include: common API header files used by several platforms
 - platform/common: common code for several platforms
 - specifics platform folders , See below the supported platform list
 - platform/./configs folder: configuration files for framework repository and other middlewares (rpmsg, mbedTls, etc..)
- Services folder
- Zephyr folder for zephyr modules integrated in mcux SDK
- clang formatting script and script folder to format appropriately the source files of the repo

Supported platforms The following devices/platforms are supported in platform folder for connectivity applications:

- kw45x, k32w1x, mcxw71x, under wireless_mcu, kw45_k32w1_mcxw71 folders.
- kw47x, mcxw72x families under wireless_mcu, kw47_mcxw72, kw47_mcxw72_nbu folders.
- rw61x
- RT1060 and RT1170 for Matter
- Other RT devices such as i.MX RT595s

Supported services The supported services are provided for connectivity stacks and their demo application, and are usually dependent on PLATFORM API implementation:

- **DBG:** Light Debug Module, currently a stubbed header file
- **FSCI:** Framework Serial Communication Interface between BLE host stack and upper layer located on an other core/device
- **FunctionLib:** wrapper to toolchain memory manipulation functions (memcpy, memcmp, etc) or use its own implementation for code size reduction
- **HWPParameters:** Store Factory hardware parameters and Application parameters in Flash or IFR
- **LowPower:** wrapper of SDK power manager for connectivity applications
- **ModuleInfo:** Store and handle connectivity component versions
- **NVM:** NXP proprietary File System used for KW45, KW47 automotive devices and RT1060/RT1170 platform for Matter
- **OtaSupport:** Handle OTA binary writes into internal or external flash.
- **SecLib and RNG:** Crypto and Random Number generator functions. It supports several ports:
 - Software algorithms
 - Secure subsystem interface to an HW enclave
 - MbedTls 2.x interface
- **Sensors:** Provides service for Battery and temperature measurements
- **SFC:** Smart Frequency Calibration to be run from KW47/MCXW71 from NBU core. Matter related modules:
- **OTW:** Over The Wire module for External Transceiver firmware update from RT platforms
- **FactoryDataProvider** to be used for Matter

Supported Zephyr modules integration in mcux SDK Connectivity framework provides integration and port layers to the following Zephyr Modules located into zephyr/subsys:

- **NVS:** Zephyr File System used by Matter and Zigbee
- **Settings:** Over layer module that allows to store keys into NVS File System used by Matter Port layer and required libraries for these zephyr modules are located in port and lib folder in zephyr directory

Connectivity framework CHANGELOG

7.1.3 mcux SDK 25.12.00 RFP

Major Changes

- **[wireless_mcu][wireless_nbu]** Replaced interrupt masking macros with static inline functions `PLATFORM_SetInterruptMask()` and `PLATFORM_ClearInterruptMask()` to ensure consistent BASEPRI value handling across all compilers. This addresses compiler-dependent behavior issues with the previous macro implementation.
- **[wireless_mcu]** Added BASEPRI-based interrupt masking in `PLATFORM_RemoteActiveRel()` to allow high-priority IRQs while ensuring only IMU0 or thread context can call this function.

- [wireless_mcu] Introduced `gPlatformUseHwParameter_d` compile flag to allow builds without `HwParameter` section. When undefined or set to 0, crystal trimming functions conditionally access `HwParameters` only when required.

Minor Changes

- [kw47_mcxw72] Introduced platform-specific library for KW47/MCXW72 platforms.
- [kw47_mcxw72] Support for additional KW47 phantoms including ZB2/ZB3/ZB6/ZB7 and Z83/Z96/Z97.
- [kw47_mcxw72] Removed `SH_MEM_TOTAL_SIZE` override as it is now automatically calculated to match `rpmsg-lite` configuration.
- [wireless_mcu] Set `RL_BUFFER_PAYLOAD_SIZE` to word-aligned value as expected by `rpmsg-lite`.
- [wireless_mcu] Added system-generated HCI vendor events capability for debug and diagnostic purposes.
- [DBG] Added debug structure transmission over HCI vendor events with `NBUDBG_ConfigureHciVendorEvent()` API to enable/disable the feature.
- [DBG] Added debug structure versioning and logging buffer address/size information to debug structure.
- [DBG] Removed 15.4 region from debug structure as not supported on KW47.
- [DBG] Added stack overflow detection for `armv8_m_mainline` architecture.
- [Platform] Simplified enablement of reset features via pin detection
 - Automatically selects `gUseResetByLvdForce_c` when `gAppForceLvdResetOnResetPinDet_d` is enabled.
 - Automatically select `gUseResetByDeepPowerDown_c` when `gAppForceDeepPowerDownResetOnResetPinDet_d` is enabled.
- [RNG] Replaced `gRngHasSecLibDependency_d` compilation switch with `gRngUseSecLib_d`.
- [mcxw23] Defined `gRngIsrPrio_c` on the preprocessor to make it global and avoid redefinition warnings.
- [mcxw23] Implemented `PLATFORM_ResetCrypto()` API called by `RNG_reinit/SecLib_reinit`.
- [mcxw23] Added support of Timer Manager timestamp with `OSTIMER`.

Bug fixes

- [wireless_mcu] Fixed race condition in `PLATFORM_RemoteActiveRel()` by adding verification loop to confirm NBU core execution before releasing power domain.
- [wireless_mcu] Added instruction synchronization barrier (`__ISB()`) after interrupt re-enable in `PLATFORM_RemoteActiveRel()` to ensure pending interrupts execute between critical sections.
- [wireless_mcu] Fixed external IO voltage isolation issue during low-power initialization - isolation is now cleared at init to ensure proper behavior.
- [wireless_mcu] Replaced spin-wait loops with event-based synchronization in NBU communication APIs. Added mutex protection to `PLATFORM_NbuApiReq()` and `PLATFORM_GetNbuInfo()` to prevent race conditions and deadlocks when multiple tasks call these APIs concurrently.
- [wireless_mcu] Fixed OSA bare metal event race condition in ICS where auto-clear event feature could cause tasks to become permanently stuck. Disabled auto-clear feature in bare metal builds and manually clear event flags after `OSA_EventWait()` returns successfully.

- [NVM] Fixed `NvIdle()` to prevent looping for more operations than the queue size.
- [NVS] Fixed blank check procedure to return false (non-blank) when checking a 0 length area.
- [NVS] Made external and internal flash ports consistent.
- [DBG] Fixed debug structure size and callback access issues - corrected memory placement overlap between `reg_info` and `assert_info`.
- [MISRA] Various MISRA compliance fixes in NVM, HWParameter, LowPower, SecLib, Platform modules and IFR offset definitions.

7.1.2 mcux SDK 25.12.00 pvw2

Major Changes

- [wireless_mcu] Reduced RPMSG buffer payload size from 496 to 270 bytes on KW43/KW47 platforms, saving 226 bytes per buffer (1808 bytes total with 4 buffers on each core). This optimization is possible as `rpmsg-lite` no longer requires buffer sizes to be powers of two.
- [configs] Introduced `RL_ALLOW_CUSTOM_SHMEM_CONFIG` flag in `rpmsg_config.h` to enable connectivity applications to use `platform_set_static_shmem_config()` and `platform_get_custom_shmem_config()`.

Minor Changes

- [wireless_mcu] Updated radio power management configuration with `PLATFORM_InitRadio()` API on kw47/mcxw72 platforms
- [DBG] Added NBU assert indication support to host with line/file info using debug structure.
- [DBG] Enhanced NBU debug framework with warning detection and notification capabilities. Extended `NBUDBG_StateCheck()` to monitor NBU warnings via `PLATFORM_IsNbuWarningSet()` with callback support for proactive warning monitoring.
- [Sensors] Added periodic temperature measurement support allowing app/host to request periodic temperature measurement.
- [Sensors] Added markdown documentation explaining periodic measurement functionality upon NBU requests.
- [SecLib_RNG] Added documentation for asynchronous seed handling using `RNG_NotifyReseedNeeded()` and SecLib implementation flavors (Software, EdgeLock, PSA Crypto, MbedTLS).
- [PSA] Simplified PSA configuration files and reduced imports/definitions for wireless MCU platforms.
- [mcxw23] Changed temperature dummy value for sensors to progress on application enablement.
- [NVS] Enhanced debug capabilities by adding `CONFIG_NVS_LOG_LEVEL` and improved LOG macros to adapt to PRINTF limitations.

Bug fixes

- [wireless_mcu] Fixed FRO32K as 32 kHz clock source with deferred OSC32K switching to improve initialization performance after warm reset.
- [wireless_mcu] Added wait loop for NBU power domain readiness in `PLATFORM_InitNbu()` to prevent race conditions when accessing NBU power domain in applications without NBU images.

- [wireless_mcu] Fixed external flash blank check procedure for LSPI external NOR Flash by correcting `PLATFORM_ExternalFlashAreaIsBlank()` to read from RAM buffer and perform erase pattern comparison in RAM with optimized 4-byte step loops.
- [NVS] Removed mflash dependency from NVS external flash port and fixed internal flash blank check of unaligned flash data.
- [mcxw23] Ensured TX power does not exceed 6dBm maximum limit.

7.1.1 mcux SDK 25.12.00 pvw1

Minor Changes

- [wireless_mcu][wireless_nbu] Migrated TSTMR implementation to use SDK `fsl_tstmr` driver for better maintainability and consistency. This migration replaces custom TSTMR register definitions with official SDK driver APIs while maintaining existing `PLATFORM_*` API compatibility.

Bug fixes

- [SecLib_RNG][mbedtls] Enhanced ECDH context preservation across low-power transitions on KW45_MCXW71 and KW47_MCXW72 platforms using export/import APIs to ensure cryptographic context is retained when hardware accelerator loses internal memory during power-down mode.
- [wireless_nbu] Fixed incorrect FRODIV values that were causing reduced peripheral clocks by updating `PLATFORM_FroDiv[]` mapping array to prevent over-division of flash APB and RF_CMC clocks.
- [rw61x/rt1060/rt1170] Added missing common files for external and OTA support on rw61x/rt1060/rt1170 platforms.

7.0.3 RFP mcux SDK 25.09.00

Major Changes

- [wireless_mcu] Replaced ICS RX linked list with message queue to eliminate memory allocation in ISR context and enable user callbacks to run in thread context where memory allocation is permitted.
- [wireless_mcu] Added HCI RX workqueue processing support to reduce ISR execution time and system impact. Feature controlled by `gPlatformHciUseWorkqueueRxProcessing_d` configuration option (enabled by default on kw45_k32w1_mcxw71 and kw47_mcxw72 platforms in freertos applications). When enabled, HCI transport processes received data in system workqueue thread, allowing user callbacks to run in thread context.
- [wireless_nbu] Introduced `PLATFORM_ConfigureSmuDmemMapping()` API to configure SMU and DMEM sharing on NBU for kw47_mcxw72 platform, using linker file symbols for correct configuration.
- [mcxw23] Implemented HCI interface using PLATFORM API as preliminary requirement for Zephyr enablement, introducing `PLATFORM_SendHciMessageAlt()` alternative API.
- [wireless_mcu][wireless_nbu] Added NBU2Host event manager for status indications to host (Information, Warning, Error) sent over RPMSG.
- [wireless_mcu] Added a call to `PLATFORM_IcsRxWorkHandler()` within `PLATFORM_NbuApiReq()` for baremetal applications to prevent potential deadlocks.

Minor Changes

- [wireless_mcu] Fixed variable underflow issue in PLATFORM_RemoteActiveRel().
- [SecLib_RNG] Fixed escaping local HashKeyBuffer address issue and added missing cast in RNG_GetTrueRandomNumber() function.
- [Common] Fixed heap memory manager return values and added missing include to fwk_freertos_utils.h.
- [rw61x] Prevented array out of bounds in PLATFORM_RegisterRtcHandle().
- [FSCI] Fixed memory leak in FSCI module.
- [NVM] Enhanced debug facilitation by restricting variable scope, assigning return statuses to variables, and fixing display format in NV_ShowFlashTable().
- [wireless_mcu] Added new chip revision A2.1 support in PLATFORM_SendChipRevision() API.
- [kw47_mcxw72] Implemented BLE BD address retrieval from IFR memory with fallback to OUI + RNG.
- [DBG] Added ThreadX support to fault handlers and reworked fault handler structure with dedicated RTOS files.
- [DBG][Common] Added NBU debug support on host side to detect faults and system errors, with debug info extraction capability (limited to MCXW72/KW47).
- [Common] Platform CMake rework and Kconfig renaming, removing unneeded checks and renaming PRJSEG platform Kconfigs to COMPONENT.
- [mcxw23] Added experimental SecLib PSA support with additional configuration for MBEDTLS_ECP_C and MBEDTLS_BIGNUM_C.
- [wireless_mcu] Cleaned CMakeLists.txt to avoid wrong inclusions of files and folders from incorrect platforms.
- [wireless_mcu][wireless_nbu] Added NBU2Host warning when 32MHz crystal is unready on low power exit.
- [wireless_mcu][ot] Introduced gPlatformUseOuiFromIfr to use OUI from IFR for the extended address (disabled by default). When enabled and IFR is not blank, copies first three bytes to OUI field of extended address, otherwise uses static OUI as fallback.
- [General] Removed useless warning about TSTMR_CLOCK_FREQUENCY_MHZ definition.
- [General] Updated framework license and SBOM for 25.09 RFP release.
- [wireless_mcu] Fixed unused variable warning when gPlatformIcsUseWorkqueueRxProcessing_d and gPlatformHciUseWorkqueueRxProcessing_d are disable

7.0.3 revB mcux SDK 25.09.00

Major Changes

- [wireless_mcu] Adjusted default value of BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0x16 to 0x10 to address stability issues observed with the previous setting. This change enhances system reliability but will reduce low-power performance.

Minor Changes (bug fixes)

- [Common] Added MDK compatibility for the errno framework header.
- [mcxw23] Implemented missing PLATFORM_OtaClearBootInterface() API.

- [mcxw23] Refactored fwk_platform.c to separate BLE-specific logic into fwk_platform_ble.c.
- [OTA] Corrected definition of gEepromParams_WriteAlignment_c flag for mcxw23
- [OTA] Enabled calling OTA_GetImgState() prior to OTA_Initialize().
- [wireless_mcu] Fixed PLATFORM_IsExternalFlashSectorBlank() to check the entire sector instead of just one page.
- [mcxw23] Added support for OTA using external flash.
- [mcxw23] Introduced PLATFORM_GetRadioIdleDuration32K() to estimate time until next radio event.
- [OTA] Removed gUseInternalStorageLink_d linker flag definition when external OTA storage is used.
- [mcxw23] Extended CopyAndReboot() to support external flash OTA.
- [wireless_mcu] Resolved counter wrap issue in PLATFORM_GetDeltaTimeStamp().
- [kw43_mcxw70] Defined LPTMR frequency constants in fwk_platform_definitions.h.
- [kw47_mcxw72] Updated shared memory allocation for RPMsg adapter.
- [mcxw23] Implement PLATFORM_IsExternalFlashBusy() API.
- [kw45_mcxw71][kw47_mcxw72] Moved RAM bank definitions from the connectivity framework to device-specific definitions.

7.0.3 revA mcux SDK 25.09.00

Major Changes

- [wireless_nbu] Enhanced XTAL32M trimming handling: updates are applied when requested by the application core and the NBU enters low-power mode, ensuring no interference from ongoing radio activity. Introduced new APIs to lock (PLATFORM_LockXtal32MTrim()) and unlock XTAL32M (PLATFORM_UnlockXtal32MTrim()) trimming updates using a counter-based mechanism. Also added a reset API (PLATFORM_ResetContext()) for platform-specific variables (currently limited to the trimming lock).
- [wireless_mcu] Introduced a new API, PLATFORM_SetLdoCoreNormalDriveVoltage(), to enable support for NBU clock frequency at 64 MHz, as required by BLE channel sounding applications.
- [wireless_mcu][wireless_nbu] Increased delayLpoCycle default from 2 to 3 to address link layer instabilities in low-power NBU use cases. Adjusted BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0x10 to 0x16 to balance power consumption and stability. □ NBU may malfunction if delayLpoCycle (or BOARD_LL_32MHz_WAKEUP_ADVANCE_HSLLOT) is set to 2 while BOARD_RADIO_DOMAIN_WAKE_UP_DELAY is 0x16.

Minor Changes (bug fixes)

- [WorkQ] Increased stack size when RNG use mbedtls port and coverage is enabled.
- [FSCI] Resolved an issue where messages remained unprocessed in the queue by ensuring OSA_EventSet() is triggered when pending messages are detected.
- [OTA] Fixed a bug in OTA_PullImageChunk() that prevented retrieval of data previously received via OTA_PushImageChunk() when still buffered in RAM during posted operations.
- [OTA] Various MISRA and coverity fixes.

- [mcxw23] Fixed an unused variable warning in PLATFORM_RegisterNbuTemperatureRequestEventCb() API.
- [SFC] Remove obsolete flag gNbuJtagCapability.
- [wireless_mcu] Introduced new API PLATFORM_GetRadioIdleDuration32K(). Deprecated PLATFORM_CheckNextBleConnectivityActivity() API.
- [mcxw23] Aligned platform-specific implementations with the corresponding prototypes defined in wireless_mcu.
- [DBG] Cleaned up fwk_fault_handler.c.

7.0.2 RFP mcux SDK 25.06.00

Major Changes

- [wireless_mcu][wireless_nbu] Introduced PLATFORM_Get32KTimeStamp() API, available on platforms that support it.
- [RNG] Switched to using a workqueue for scheduling seed generation tasks.
- [Sensors] Integrated workqueue to trigger temperature readings on periodic timer expirations.
- [wireless_nbu] Removed outdated configuration files from wireless_nbu/configs.
- [SecLib_RNG][PSA] Added a PSA-compliant implementation for SecLib_RNG. □ This is an experimental feature and should be used with caution.
- [wireless_mcu][wireless_nbu] Implemented PLATFORM_SendNBUXtal32MTrim() API to transmit XTAL32M trimming values to the NBU.

Minor Changes (bug fixes)

- [MWS] Migrated the Mobile Wireless Standard (MWS) service to the public repository. This service manages coexistence between connectivity protocols such as BLE, 802.15.4, and GenFSK.
- [HWPParameter][NVM][SecLib_RNG][Sensors] Addressed various MISRA compliance issues across multiple modules.
- [Sensors] Applied a filtering mechanism to temperature data measured by the application core before forwarding it to the NBU, improving data reliability.
- [Common] Relocated the GetPowerOfTwoShift() function to a shared module for broader accessibility across components.
- [RNG] Resolved inconsistencies in RNG behavior when using the fsl_adapter_rng HAL by aligning it with other API implementations.
- [SecLib] Updated the AES CMAC block counter in AES_128_CMAC() and AES_128_CMAC_LsbFirstInput() to support data segments larger than 4KB.
- [SecLib] Utilized sss_sscp_key_object_free() with kSSS_keyObjFree_KeysStoreDefragment to avoid key allocation failures.
- [MCXW23] Removed redundant NVIC_SetPriority() call for the ctimer IRQ in the platform file, as it's already handled by the driver.
- [WorkQ] Increased workqueue stack size to accommodate RNG usage with mbedtls.
- [wireless_mcu][ot] Suppressed chip revision transmission when operating with nbu_15_4.
- [platform][mflash] Ensured proper address alignment for external flash reads in PLATFORM_ReadExternalFlash() when required by platform constraints.

- [RNG] Corrected reseed flag behavior in `RNG_GetPseudoRandomData()` after reaching `gRng-MaxRequests_d` threshold.
- [platform][mflash] Fixed uninitialized variable issue in `PLATFORM_ReadExternalFlash()`.
- [platform][wireless_nbu] Fixed an issue on KW47 where `PLATFORM_InitFro192M` incorrectly reads IFR1 from a hardcoded flash address (0x48000), leading to unstable FRO192M trimming. The function is now conditionally compiled for KW45 only.

7.0.2 revB mcux SDK 25.06.00

Major Changes

- [RNG][wireless_mcu][wireless_nbu] Rework RNG seeding on NBU request
- [wireless_mcu] [LowPower] Add `gPlatformEnableFro6MCalLowpower_d` macro to enable FRO6M frequency verification on exit of Low Power
 - add `PLATFORM_StartFro6MCalibration()` and `PLATFORM_EndFro6MCalibration()` new function for FRO6M calibration (6MHz or 2Mhz) on wake-up from low power mode.
 - Enabled by default in `fwk_config.h`
- [wireless_nbu][LowPower] Clear pending interrupt status of the systick before going in low-power - Reduce NBU active time
- [wireless_nbu] Fix impossibility to go to WFI in combo mode (15.4/BLE)
- [wireless_mcu] Implement XTAL32M temperature compensation mechanism. 2 new APIs:
 - `PLATFORM_RegisterXtal32MTempCompLut()`: register the temperature compensation table for XTAL32M.
 - `PLATFORM_CalibrateXtal32M()`: apply XTAL32M temperature compensation depending on current temperature.
- [Sensors][wireless_mcu] Add support for periodic temperature measurement. new API:
 - `SENSORS_TriggerTemperatureMeasurementUnsafe()`: to be called from Interrupt masked critical section, from ISR or when scheduler is stopped
- [SFC] Change default maximal ppm target of the SFC algorithm from 200 to 360ppm. Impact the SFC algorithm of kw45 and mcxw71 platforms, 360ppm was already the default setting for kw47 and mcxw72 platforms

Minor Changes (bug fixes)

- [DBG] Fix `FWK_DBG_PERF_DWT_CYCLE_CNT_STOP` macro
- [wireless_nbu] Add `gPlatformIsNbu_d` compile Macro set to 1
- [wireless_nbu][ics] `gFwkSrvHostChipRevision_c` can be processed in the system workqueue
- [kw45_mcxw71][kw47_mcxw72]
 - Remove LTC dependency from platform in `kconfig`
 - `gPlatformShutdownEccRamInLowPower` moved from `fwk_platform_definition.h` to `fwk_config.h` as this is a configuration flag.
- [wireless_mcu][sensors] Rework and remove unnecessary ADC APIs
- [wireless_nbu] Add `PLATFORM_GetMCUUid()` function from Chip UID
- [SecLib] Change `AES_MMO_BlockUpdate()` function from private to public for zigbee.

7.0.2 revA mcux SDK 25.06.00 Supported platforms:

- Same as 25.03.00 release

Major Changes

- [KW45/MCXW71] HW parameters placement now located in IFR section. Flash storage is not longer used:
 - Compilation: Macro `gHwParamsProdDataPlacement__c` changed from `gHwParamsProdDataMainFlash2IfcMode__c` to `gHwParamsProdDataIfcMode__c`
- [KW47] NBU: Add new `fwk_platform_dcdc.[ch]` files to allow DCDC stepping by using SPC high power mode. This requires new API in `board_dcdc.c` files. Please refer to new compilation MACROs `gBoardDcdcRampTrim__c` and `gBoardDcdcEnableHighPowerModeOnNbu__d` in `board_platform.h` files located in `kw47evk`, `kw47loc`, `frdm-mcxw72` board folders.
- [KW45/MCXW71/KW47/MCXW72] Trigger an interrupt each time App core calls `PLATFORM_RemoteActiveReq()` to access NBU power domain in order to restart NBU core for domain low power process

Minor Changes (bug fixes)

Services

- [SecLib_RNG]
 - Rename `mSecLibMutexId` mutex to `mSecLibSssMutexId` in `SecLib_sss.c`
 - Remove `MEM_TRACKING` flag from `RNG.c`
 - Implement port to `fsl_adapter_rng.h` API using `gRngUseRngAdapter__c` compil Macro from `RNG.c`
 - Add support for BLE debug Keys in `SecLi` and `SecLin_sss.c` with `gSecLibUseBleDebugKeys__d` - for Debug only
- [FSCI] Add queue mechanism to prevent corruption of FSCI global variable. Allow the application to override the trig sample number parameter when `gFsciOverRpmMsg__c` is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using `SBT-SNOOP_MODE_RAW`
- [OTA]
 - `OtaInternalFlash.c`: Take into account chunks smaller than a flash phrase worth
 - `fwk_platform_ot.c`: dependencies and include files to `gpio`, `port`, `pin_mux` removed

Platform specific

- [kw45-mcxw71][kw47-mcxw72]
 - `fwk_platform_reset.h` : add compil Macro `gUseResetByLvdForce__c` and `gUseResetByDeepPowerDown__c` to avoid compile the code if not supported on some platforms
 - New compile Flag `gPlatformHasNbu__d`
 - Rework FRO32K notification service for MISRA fix

7.0.1 RFP mcux SDK 25.03.00 Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, SecLib and platform files

Services

- [SecLib_RNG] fix return status from RNG_GetTrueRandomNumber() function: return correctly gRngSuccess_d when RNG_entropy_func() function is successful
- [SFC] Allow the application to override the trig sample number parameter
- [Settings] Re-define the framework settings API name to avoid double definition when gSettingsRedefineApiName_c flag is defined

Platform specific

- [wireless_mcu] fwk_platform_sensors update :
 - Enable temperature measurement over ADC ISR
 - Enable temperature handling requested by NBU
- [wireless_mcu] fwk_platform_lcl coex config update for KW45
- [kw47_mcxw72] Change the default ppm_target of SFC algorithm from 200 to 360ppm

7.0.1 revB mcux SDK 25.03.00 Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

General

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, FunctionLib and platform files

Services

- [SecLib_RNG] AES-CBC evolution:
 - added AES_CBC_Decrypt() API for sw, SSS and mbedtls variants.
 - Made AES-CBC SW implementation reentrant avoiding use of static storage of AES block.
 - fixed SSS version to update Initialization Vector within SecLib, simplifying caller's implementation.
 - modified AES_128_CBC_Encrypt_And_Pad() so as to avoid the constraint mandating that 16 byte headroom be available at end of input buffer.
- [SecLib_RNG] RNG modifications:
 - RNG_GetPseudoRandomData() could return 0 in some error cases where caller expected a negative status.
 - * Explicated RNG error codes
 - * Added argument checks for all APIs and return gRngBadArguments_d (-2) when wrong
 - * added checks of RNG initialization and return gRngNotInitialized_d (-3) when not done
 - * fixed correctness of RNG_GetPrngFunc() and RNG_GetPrngContext() relative to API description.
 - * Added RNG_DeInit() function mostly for test and coverage purposes.
 - * Improved RNG description in README.md
 - * Unified the APIs behaviour between mbedtls and non mbedtls variants.
 - RNG/mbedtls: Prevent RNG_Init() from corrupting RNG entropy context if called more than once.
 - RNG/mbedtls: fixed RNG_GetTrueRandomNumber() to return a proper mbedtls_entropy_func() result.
 - Use defragmentation option when freeing key object in SecLib_sss to avoid leak in S200 memory
 - Add new API ECP256_IsKeyValid() to check whether a public key is valid
- [OtaSupport] Update return status to OTA_Flash_Success when success at the end of InternalFlash_WriteData() and InternalFlash_FlushWriteBuffer() APIs
- [WorQ] Implementing a simple workqueue service to the framework
- [SFC] Keep using immediate measurement for some measurement before switching to configuration trig to confirm the calibration made
- [DBG] Adding modules to framework DBG :
 - sbtsnoop
 - SWO
- [Common] Fix HAL_CTZ and HAL_RBIT IAR versions
- [LowPower] Fix wrong tick error calculation in case of infinite timeout
- [Settings] Add new macro gSettingsRedefineApiName_c to avoid multiple definition of settings API when using connectivity framework repo

Platform specific

- [KW47/MCXW72] Change xtal cload default value from 4 to 8 in order to increase the precision of the link layer timebase in NBU
- [wireless_mcu] [wireless_nbu] Use new WorkQ service to process framework intercore messages
- [rw61x] Fix HCI message sending failure in some corner case by releasing controller wakes up after that the host has send its HCI message
- [MCXW23] Adding the initial support of MCXW23 into the framework

7.0.0 mcux SDK 24.12.00 Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170

Minor Changes (bug fixes)

Platform specific

- [RW61X]
 - Add MCUX_COMPONENT_middleware.wireless.framework.platform.rng to the platform to fix a warning at generation
 - Retrieve IEEE 64 bits address from OTP memory
- [KW45x, MCXW71x, KW47x, MCXW72x]
 - Ignore the secure bit from RAM addresses when comparing used ram bank in bank retention mechanism
 - Add gPlatformNbuDebugGpioDAccessEnabled_d Compile Macro (enabled by default). Can be used to disable the NBU debug capability using IOs in case Trustzone is enabled (“PLATFORM_InitNbu()’ code executed from unsecure world).
 - Fix in NBU firmware when sending ICS messages gFwkSrvNbuApiRequest_c (from controller_api.h API functions)

Services

- [OTA]
 - Add choice name to OtaSupport flash selection in Kconfig
- [NVM]
 - Add gNvmErasePartitionWhenFlashing_c feature support to gcc toolchain
- [SecLib_RNG]
 - Misra fixes

7.0.0 revB mcux SDK 24.12.00 Supported platforms: KW45x, KW47x, MCXW71, MCXW72, K32W1x, RW61x, RT595, RT1060, RT1170

Major Changes (User Applications may be impacted)

- mcux github support with cmake/Kconfig from sdk3 user shall now use CmakeLists.txt and Kconfig files from root folder. Compilation should be done using west build command. In order to see the Framework Kconfig, use command >west build -t guiconfig
- Board files and linker scripts moved to examples repository

Bugfixes

- [platform lowpower]
 - Entering Deep down power mode will no longer call PLATFORM_EnterPowerDown(). This API is now called only when going to Power down mode

Platform specific

- [KW47/MCXW72]: Early access release only
 - Deep sleep power mode not fully tested. User can experiment deep sleep and deep down modes using low power reference design applications
 - XTAL32K-less support using FRO32K not tested
- [KW45/MCXW71/K32W148]
 - Deep sleep mode is supported. Power down mode is supported in low power reference design applications as experimental only
 - XTAL32K-less support using FRO32K is experimental - FRO32K notifications callback is debug only and should not be used for mass production firmware builds

Minor Changes (no impact on application)

- Overall folder restructuring for SDK3
 - [Platform]:
 - * Rename platform_family from connected_mcu/nbu to wireless_mcu/nbu
 - * platform family have now a dedicated fwk_config.h, rpmsg_config.h and SecLib_mbedtls_config.h
 - [Services]
 - * Move all framework services in a common directory “services/”

7.0.0 revA: KW45/KW47/MCX W71/MCX W72/K32W148

Experimental Features only

- Power down on application power domain: Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support: Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as PRINTF) as it is executed in ISR context.

Main Changes

- Cmake/Kconfig support for SDK3.0
- [Sensors] API renaming:
 - `SENSORS__InitAdc()` renamed to `SENSORS__Init()`
 - `SENSORS__DeinitAdc()` renamed to `SENSORS__Deinit()`
- [HWparams]
 - Repair PROD_DATA sector in case of ECC error (implies loss of previous contents of sector)
- [NVM] Linker script modification for armgcc whenever `gNvTableKeptInRam_d` option is used:
 - placement of `NVM_TABLE_RW` in data initialized section, providing start and end address symbols. For details see `NVM_Interface.h` comments.
- [OtaSupport]
 - `OTA__Initialize()`: now transitions the image state from RunCandidate to Permanent if not done by the application. OTA module shall always be initialized on a Permanent image, this change ensures it is the case.
 - `OTA__MakeHeadRoomForNextBlock()`: now erases the OTA partition up to the image total size (rounded to the sector) if known.

Minor changes

- [Platform]
 - Updated macro values: -kw47: `BOARD_32MHZ_XTAL_CDAC_VALUE` from 12U to 16U, `BOARD_32MHZ_XTAL_ISEL_VALUE` from 7U to 11U, `BOARD_32KHZ_XTAL_CLOAD_DEFAULT` from 8U to 4U, `BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT` from 1U to 3U
 - * MCX W72 (low-power reference design applications only): `BOARD_32MHZ_XTAL_CDAC_VALUE` from 12U to 10U, `BOARD_32MHZ_XTAL_ISEL_VALUE` from 7U to 11U, `BOARD_32KHZ_XTAL_CLOAD_DEFAULT` from 8U to 4U, `BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT` from 1U to 3U
 - New `PLATFORM__RegisterNbuTemperatureRequestEventCb()` API: register a function callback when NBU request new temperature measurement. API provides the interval request for the temperature measurement
 - Update `PLATFORM__IsNbuStarted()` API to return true only if the NBU firmware has been started.
- [platform lowpower]
 - Move RAM layout values in `fwk_platform_definition.h` and update RAM retention API for KW47/MCXW72

Bugfixes

- [OtaSupport]
 - `OTA__MakeHeadRoomForNextBlock()`: fixed a case where the function could try to erase outside the OTA partition range.

6.2.4: KW45/K32W1x/MCXW71/RX61x SDK 2.16.100 This release does not contain the changes from 6.2.3 release.

This release contains changes from 6.2.2 release.

Main Change

- armgcc support for Cmake sdk2 support and VS code integration

Minor changes

- [NBU]
 - Optimize some critical sections on nbu firmware
- [Platform]
 - Optimize PLATFORM_RemoteActiveReq() execution time.

6.2.3: KW47 EAR1.0 Initial Connectivity Framework enablement for KW47 EAR1.0 support.

New features

- OpenNBU feature : nbu_ble project is available for modification and building

Supported features

- Deep sleep mode

Unsupported features

- Power down mode
- FRO32K support (XTAL32K less boards)

Main changes

- [NBU]
 - LPTMR2 available and TimerManager initialization with Compile Macro: gPlatformUseLptmr_d
 - NBU can now have access to GPIOD
 - SW RNG and SW SecLib ported to NBU (Software implementation only)
- [RNG]
 - Obsoleted API removed : FWK_RNG_DEPRECATED_API
 - RNG can be built without SecLib for NBU, using gRngUseSecLib_d in fwk_config.h
 - Some API updates:
 - * RNG_IsReseedneeded() renamed to RNG_IsReseedNeeded,
 - * RNG_TriggerReseed() renamed to RNG_NotifyReseedNeeded(),
 - * RNG_SetSeed() and RNG_SetExternalSeed() return status code.
 - Optimized Linear Congruential modulus computation to reduce cycle count.

Minor changes

- [NVM]
 - Optimize `NvIsRecordErased()` procedure for faster garbage collection
 - MISRA fix : Remove externs and weaks from NVM module - Make RNG and timer manager dependencies conditional
- [Platform]
 - Allow the debugger to wakeup the KW47/MCXW72 target

6.2.2: KW45/K32W1 MR6 SDK 2.16.000 Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as `PRINTF`) as it is executed in ISR context.

Changes

- [Board] Support for freedom board FRDM-MCX W7X
- [HWparams]
 - Support for location of `HWParameters` and `Application Factory Data IFR` in `IFR1`
 - Default is still to use `HWparams` in Flash to keep backward compatibility
- [RNG]: API updates:
 - New APIs `RNG_IsReseedneeded()`, `RNG_SetSeed()` to provide Seed to PRNG on NBU/App core - See `BluetoothLEHost_ProcessIdleTask()` in `app_conn.c`
 - New APIs `RNG_SetExternalSeed()` : User can provide external seed. Typically used on NBU firmware for App core to set a seed to RNG. `RNG_TriggerReseed()` : Not required on App core. Used on NBU only.
- [NVS] Wear statistics counters added - Fix `nvs_file_stat()` function
- [NVM] fix `Nv_Shutdown()` API
- [SecLib] New feature AES MMO supported for Zigbee

6.2.2: RW61x RFP4 SDK 2.16.000

- [Platform] Support Zigbee stack
- [OTA] Add support for RW61x OTA with remap feature.
 - Required modifications to prevent direct access to flash logical addresses when remap is active.
 - Image trailers expected at different offset with remap enabled (see `gPlatformMcuBootUseRemap_d` in `fwk_config.h`)
 - fixed image state assessment procedure when in `RunCandidate`.
- [NVS] Wear statistics counters added
- [SecLib] New feature AES MMO supported for Zigbee
- [Misra] various fixes

6.2.1: KW45/K32W1 MR5 SDK 2.15.000 Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress. Timing variation of the timebase are being analyzed

Major changes

- [RNG]: API updates
 - New compile flag to keep deprecated API: FWK_RNG_DEPRECATED_API
 - change return error code to int type for RNG_Init(), RNG_ReInit()
 - New APIs RNG_GetTrueRandomNumber(), RNG_GetPseudoRandomData()
- [Platform]
 - fwk_platform_sensors
 - * Change default temperature value from -1 to 999999 when unknown
 - fwk_platform_genfsk
 - * rename from platform_genfsk.c/h to fwk_platform_genfsk.c/h
 - platform family
 - * Rename the framework platform folder from kw45_k32w1 to connected_mcu to support other platform from the same family
 - fwk_platform_intflash
 - * Moved from fwk_platform files to the new fwk_platform_intflash files the internal flash dependant API
- [NBU]
 - BOARD_LL_32MHz_WAKEUP_ADVANCE_HSL0T changed from 2 to 3 by default
 - BOARD_RADIO_DOMAIN_WAKE_UP_DELAY changed from 0x10 to 0x0F
- [gcc linker]
 - Exclude k32w1_nbu_ble_15_4_dyn.bin from .data section

Minor Changes

- [Platform]
 - PLATFORM_GetTimeStamp() has an important fix for reading the Timestamp in TSTMRO
 - New API PLATFORM_TerminateCrypto(), PLATFORM_ResetCrypto() called from SecLib for lowpower exit
 - Fix when enable fro debug callback on nbu
- [DBG]
 - SWO
 - * Add new files fwk_debug_swo.c/h to use SWO for debug purpose
 - * Two new flags has been added:
 - BOARD_DBG_SWO_CORE_FUNNEL to chose on which core you want to use SWO

- BOARD_DBG_SWO_PIN_ENABLE to enable SWO on a pin
- [NVS]
 - Add support of NVS and Settings in framework
- [NBU]
 - Fix power down issues and reduce critical section on NBU side:
 - * new API PLATFORM_RemoteActiveReqWithoutDelay() called from NBU functions where waiting delay is not required
 - * Increase delay needed in power down for OEM part to request the SOC to be active
 - * Remove unnecessary code to PLATFORM_RemoteActiveReqWithoutDelay() from PLATFORM_HciRpmMsgRxCallback()
 - * Improve nbu memory allocation failure debug messages
- [SDK]
 - Multicore: remove critical section in HAL_RpmMsgSendTimeout() (only required in FPGA HDI mode)
 - Flash drivers: update for ECC detection
- [Platform]
 - fwk_platform_sensors
 - * Fix temperature reporting to NBU
 - fwk_platform_extflash
 - * Align .c and .h prototype of PLATFORM_ExternalFlashAreaIsBlank() function
- [NVM]
 - Keep Mutex in NvModuleDeInit(). In Bare metal OS, Mutex can not be destroyed
 - New API NvRegisterEccFaultNotificationCb() to register Notification callback when Ecc error happens in FileSystem
- [MISRA] fixes
 - SecLib_sss.c: ECDH_P256_ComputeDhKey()
 - fwk_platform_extflash.c: PLATFORM_IsExternalFlashPageBlank()
 - fwk_fs_abstraction.c: Various fixes
- [HWparams]
 - Fix on if condition when gHwParamsProdDataPlacementLegacy2IfrMode_c mode is selected
- [OTA]
 - Enable gOtaCheckEccFaults_d by default to avoid bus in case of ECC error during OTA
 - Fix OTA partition overflow during OTA stop and resume transfer
- [BOARD]
 - Place code button or led specific under correct defines in board_comp.c/h
 - Bring back MACROS BOARD_INITRFSWITCHCONTROL Pins in pin_mux header file of the loc board
- [SecLib]
 - Add some undefinition in SecLib_mbedtls_config as new dependency has been added in mbedtls repo:

- * MBEDTLS_SSL_CBC_RECORD_SPLITTING, MBEDTLS_SSL_PROTO_TLS1, MBEDTLS_SSL_PROTO_TLS1_1
- [FRO32K]
 - FRO32K notification callback PLATFORM_FroDebugCallback_t() has new parameter to report the fro_trim value
 - maxCalibrationIntervalMs value can be provided to NBU using PLATFORM_FwkSrvSetRfSfcConfig()
- [Sensors]
 - fix: PLATFORM_GetTemperatureValue() shall have NBU started to send temperature to NBU

6.2.1: RW61x RFP3

- [NVS]
 - Add support of NVS and Settings in framework
- [MISRA] fixes
 - board_lp.c BOARD_UninitDebugConsole() and BOARD_ReinitDebugConsole()
 - fwk_platform_ble.c: Various fixes
- [OTA]
 - Fix OTA partition overflow during OTA stop and resume transfer

6.2.0: RT1060/RT1170 SDK2.15 Major

6.1.8: KW45/K32W1 MR4

- [BOARD PLATFORM]
 - Move gBoardUseFro32k_d to board_platform.h file
 - Offer the possibility to change the source clock accuracy to gain in power consumption
- [BOARD LP]
 - Move PLATFORM_SetRamBanksRetained() at end of BOARD_EnterLowPowerCb() in case a memory allocation is done previously in this function
 - fix low power; increase BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0 to 0x10 - Skip this delay when App requesting NBU wakeup
- [PLATFORM]
 - fwk_platform_ble.c/h: New timestamp API that returns the difference between the current value of the LL clock and the argument of the function
 - fwk_platform.c/h:
 - * New PLATFORM_EnableEccFaultsAPI_d compile flag: Enable APIs for interception of ECC Fault in bus fault handler
 - * New gInterceptEccBusFaults_d compile flag: Provide FaultRecovery() demo code for bus fault handler to Intercept bus fault from Flash Ecc error
- [LOC]
 - Incorrect behavior for set_dtest_page (DqTEST11 overridden)
 - Fix SW1 button wake able on Localization board

- Fix yellow led not properly initialized
- Format localization pin_mux.c/h files
- [Inter Core]
 - Affect values to enumeration giving the inter core service message ids
 - Shared memory settings shared between both cores
 - Add callback to register when NBU has unrecoverable Radio issue
- [NVM]
 - Add NV_STORAGE_MAX_SECTORS, NV_STORAGE_SIZE as linker symbol for alignment with other toolchain
 - ECC detection and recovery. New gNvSalvageFromEccFault_d and gNvVerifyReadBackAfterProgram_d compile flags. Please refer to ECC Fault detection section in README.md file located in NVM folder
- [OTA]
 - Prevent bus fault in case of ECC error when reading back OTA_CFR update status (disable by default)
- [SecLib]
 - Shared mutex for RNG and SecLib as they share same hardware resource
- [Key storage]
 - Fix to ignore the garbage at the end of buffers
 - Detect when buffers are too small in KS_AddKey() functions
- [FileCache]
 - Fix deadlock in Filecache FC_Process()
- [SDK]
 - Applications: remove definition of stack location and use default from linker script, fix warmboot stack in freertos at 0x20004000
 - Memory Manager Light:
 - * fix Null pointer harfault when MEM_STATISTICS_INTERNAL enable
 - * Fix MemReinitBank() on wakeup from lowpower when Ecc banks are turned off

6.1.7: KW45/K32W1 MR3

- [OTA]
 - New API OTA_SetNewImageFlagWithOffset()
 - Fix StorageBitmapSize calculation
 - OTA clean up: Removed OTA_ValidateImage()
- [Low Power]
 - New linker Symbol m_lowpower_flag_start in linker file.
 - * Flag is used to indicate NBU that Application domain goes to power down mode. Keep this flag to 0 if only Deep sleep is supported
 - * This flag will be set to 1 if Application domain goes to power down mode
 - Re-introduce PWR-AllowDeviceToSleep()/PWR-DisallowDeviceToSleep(), PWR_IsDeviceAllowedToSleep() API

- Implement tick compensation mechanism for idle hook in a dedicated freertos utils file `fwk_freertos_utils[ch]`, new functions: `FWK_PreIdleHookTickCompensation()` and `FWK_PostIdleHookTickCompensation()`
- Rework timestamping on K4W1
 - * `PLATFORM_GetMaxTimeStamp()` based on TSTMR
 - * Rename `PLATFORM_GetTimeStamp()` to `PLATFORM_GetTimeStamps()`
 - * Update `PLATFORM_Delay()`: Rework to use TSTMR instead of LPTMR for platform_delay
 - * Update `PLATFORM_WaitTimeout()`: Fixed a bug in `PLATFORM_WaitTimeout()` related to timer wrap
 - * Add `PLATFORM_IsTimeoutExpired()` API
- Fix race condition in `PWR_EnterLowPower()`, masking interrupts in case not done at upper layer
- Low power timer split in new files `fwk_platform_lowpower_timer[ch]`
- New `PWR_systicks_bm.c` file for bare metal usage: implement SysTick suspend/resume functionality, New weak `PWR_SysTicksLowPowerInit()`
- [FRO32K]
 - Improve FRO32K calibration in NBU
 - create `PLATFORM_InitFro32K()` to initialize FRO32K instead of XTAL32K (to be called from `hardware_init()`)
 - update FRO32K README.md file in SFC module
 - Debug:
 - Add Notification callback feature for SFC module FRO32K
 - Linker script update to support `m_sfc_log_start` in SMU2
- [SecLib]
 - Remove `gSecLibSssUseEncryptedKeys_d` compile option, split Secure/Unsecure APIs
 - RNG update to use same mutex than SecLib
 - Fix `AES_128_CBC_Encrypt_And_Pad` length
 - Implement `RNG_ReInit()` for lowpower
 - Fix issue in `ECDH_P256_GenerateKeys()` when waking up from power down
 - Call `CRYPTO_ELEMU_reset()` from `SecLib_reInit()` for power down support
- [BOARD]
 - Create new `board_platform.h` file for all Board characteristics settings (32Mhz XTAL, 32KHZ XTAL, etc..)
 - `TM_EnterLowpower()` to be called from LP callbacks
 - Support Localization boards, Only BUTTON0 supported
 - * New compile flag `BOARD_LOCALIZATION_REVISION_SUPPORT`
 - * New `pin_mux[ch]` files
 - Offer the possibility to override CDAC and ISEL 32MHz settings before the initialization of the crystal in `board_platform.h`
 - * new `BOARD_32MHZ_XTAL_CDAC_VALUE`, `BOARD_32MHZ_XTAL_ISEL_VALUE`
 - * `BOARD_32MHZ_XTAL_TRIM_DEFAULT` obsoleted

- [NVM file system]
 - Look ahead in pending save queue - Avoid consuming space to save outdated record
 - Fix NVM gNvDualImageSupport feature in NvIsRecordCopied
- [Inter Core]
 - Change PLATFORM_NbuApiReq() API return parameters granularity from uint32 to uint8
 - MAX_VARIANT_SZ change from 20 to 25
 - Set lp wakeup delay to 0 to reduce time of execution on host side, NBU waits XTAL to be ready before starting execution
 - Update inter core config rpmsg_config.h
 - Add timeout to while loops that relies on hardware in RemoteActiveReq(), Application can register Callbacks when timeout
 - Return non-0 status when calling PLATFORM_FwkSrvSendPacket when NBU non started
 - Let PLATFORM_GetNbuInfo return -10 if response not received on timeout - Doxygen platform_ics APIs
- [HW params]
 - New compile Macro for HW params placement in IFR - Save 8K in FLash: gHwParamsProdDataPlacement_c . 3 modes:
 - Legacy placement, move from legacy to IFR, IFR only placement
 - New compile Macro for Application data to be stored with HW params (in shared flash sector): gHwParamsAppFactoryDataExtension_d, New APIs:
 - * Nv_WriteAppFactoryData(), Nv_GetAppFactoryData()
 - See HWParameter.h
- [Platform]
 - Implement PLATFORM_GetIeee802_15_4Addr() API in fwk_platform_ot.c - New gPlatformUseUniqueIdFor15_4Addr_d compile Macro
 - Wakeup NBU domain when reading RADIO_CTRL UID_LSB register in PLATFORM_GenerateNewBDAddr()
- [Reset]
 - New reset Implementations using Deep power down mode or LVD:
 - * new files fwk_platform_reset.[ch]
 - * new APIs: PLATFORM_ForceDeepPowerDownReset(), PLATFORM_ForceLvdReset() + reset on ext pins
 - * new compile flags: gAppForceDeepPowerDownResetOnResetPinDet_d and gAppForceLvdResetOnResetPinDet_d to reset on external pins
- [FSCI]
 - fix when gFsciRxAck_c enabled
 - integrate new reset APIs

6.1.4: RW610/RW612 RFP1

- [Low Power]
 - Added support of low power for OpenThread stack.
 - Added PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep/PWR_IsDeviceAllowedToSleep APIs.
- [platform]
 - Added PLATFORM_GetMaxTimeStamp API.
 - Fixed high impact Coverity.
- [FreeRTOS]
 - Created a new utilities module for FreeRTOS: fwk_freertos_utils.c/h.
 - Implemented a tick compensation mechanism to be used in FreeRTOS idle hook, likely around flash operations. This mechanism aims to estimate the number of ticks missed by FreeRTOS in case the interrupts are masked for a long time.

6.1.4: KW45/K32W1 MR2

- [Low power]
 - Powerdown mode tested and enabled on Low Power Reference Design applications
 - XTAL32K removal functionality using FRO32K, supported from NBU firmwares - limitation: Application domain supports Deep Sleep only (not power down)
 - NBU low power improvement: low power entry sequence improvement and system clock reduction to 16Mhz during WFI
 - Wake up time from cold boot, reset, power switch greatly improved. Device starts on FRO32K, switch to XTAL32K when ready if gBoardUseFro32k_d not set
 - Bug fixes:
 - * Move PWR LowPower callback to PLATFORM layers
 - * Fix wrong compensation of SysTicks
 - * Reinit system clocks when exiting power down mode: BOARD_ExitPowerDownCb(), restore 96MHz clock is set before going to low power
 - * Call Timermanager lowpower entry exit callbacks from PLATFORM_EnterLowPower()
 - * Update PLATFORM_ShutdownRadio() function to force NBU for Deep power down mode
 - K32W1:
 - * Support lowpower mode for 15.4 stacks
- [NVM]
 - New Compilation MACRO gNvDualImageSupport to support multiple firmware image with different register dataset
 - Change default configuration gNvStorageIncluded_d to 1, gNvFragmentation_Enabled_d to 1, gUnmirroredFeatureSet_d to TRUE
 - Some MISRA issues for this new configuration.
 - Remove deprecated functionality gNvUseFlexNVM_d
- [SecLib]

- New NXP Ultrafast ecp256 security library:
 - * New optimized API for ecdh DhKey/ecp256 key pair computation: Ecdh_ComputeDhKeyUltraFast(), ECP256_GenerateKeyPairUltraFast().
 - * New macro gSecLibUseDspExtension_d.
 - * Improved software version of Seclib with Ultrafast library for ECP256_LePointValid()
- Bug fixes:
 - * Share same mutex between Seclib and RNG to prevent concurrent access to S200
 - * Optimized S200 re-initialization, restore ecdh key pair after power down
 - * Fixed race condition when power down low power entry is aborted
 - * Endianness function updates and clean up
- [OTA]
 - OTASupport improvements:
 - * New API OTA_GetImgState(), OTA_UpdateImgState()
 - * OTASupport and fwk_platform_extflash API updates for external flash: OTA_SelectExternalStoragePartition(), PLATFORM_IsExternalFlashSectorBlank(), PLATFORM_IsExternalFlashPageBlank(), PLATFORM_OtaGetOtaPartitionConfig()
 - * Updated OtaExternalFlash.c, 2 new APIs in fwk_platform_extflash.c
 - * Removed unused FLASH_op_type and FLASH_TransactionOpNode_t definitions from public API
 - * Removed unused InternalFlash_EraseBlock() from OtaInternalFlash.c
- [NBU firmware]
 - Mechanism to set frequency constraint to controller from the host PLATFORM_SetNbuConstraintFrequency()
 - NbuInfo has one more digit in versionBuildNo field
- [Board]
 - Support Extflash low power mode, add BOARD_UninitExternalFlash(), PLATFORM_UninitExternalFlash(), PLATFORM_ReinitExternalFlash()
 - Support XTAL32K removal functionality, use FRO32K instead by setting gBoardUseFro32k_d to 1 in board.h file
 - Support localization boards KW45B41Z-LOC Rev C
 - Low power improvement: New BOARD_InitPins() and BOARD_InitPinButtonBootConfig() called from hardware_init.c
 - Removed KW45_A0_SUPPORT support (dcdc)
 - Bug fixes:
 - * Fixed glitches on the serial manager RX when exiting from power down
 - * Fixed ADC not deinitialized in clock gated modes in BOARD_EnterLowPowerCb()
 - * Fixed UART output flush when going to low power: BOARD_UninitAppConsole()
- [platform]
 - PLATFORM_InitBle(), PLATFORM_SendHci() can now block with timeout if NBU does not answer. Application can register callback function to be notified when it occurs: PLATFORM_RegisterBleErrorCallback()

- Added API to set and get 32Khz XTAL capacitance values: PLATFORM_GetOscCap32KValue() and PLATFORM_SetOscCap32KValue()
- Added new Service FWK call gFwkSrvNbuMemFullIndication_c to get NBU mem full indication, register with PLATFORM_RegisterNbuMemErrorCallback()
- Added support negative value in platform intercore service
- [linker script]
 - Realigned gcc linker script with IAR linker script.
 - Added possibility to redefine cstack_start position
 - Added Possibility to change gNvmSectors in gcc linker script
 - Added dedicated reserved Section in shared memory for LL debugging
- [FreeRTOSConfig.h]
 - Removed unused MACRO configFRTOS_MEMORY_SCHEME and configTOTAL_HEAP_SIZE
- [HW Param]
 - Added xtalCap32K field to store XTAL32K trimming value
- [fwk_hal_macros.h]
 - Added MACRO for KB, MB and set, clear bits in bit fields
- [Debug]
 - Added MACROs for performance measurement using DWT: DBG_PERF_MEAS

6.1.3 KW45 MR1 QP1

- [Initialization] Delay the switch to XTAL32K source clock until the BLE host stack is initialized
- [lowpower] NBU wakeup from lowpower: configuration can now be programmed with BOARD_NBU_WAKEUP_DELAY_LPO_CYCLE, BOARD_RADIO_DOMAIN_WAKE_UP_DELAY in board.h file
- [NBU firmware] Major fix for NBU system clock accuracy
- [clock_config]
 - Update SRAM margin and flash config when switching system frequency
 - Trim FIRC in HSRUN case
- [XTAL 32K trim] XTAL 32K configuration can be tuned in board.h file with BOARD_32MHZ_XTAL_TRIM_DEFAULT, BOARD_32KHZ_XTAL_CLOAD_DEFAULT, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT
- [MAC address] Add OUI field in PLATFORM_GenerateNewBDAddr() when using Unique Device Id

6.1.2: RW610/RW612 PRC1

- [Low Power]
 - Updates after SDK Power Manager files renaming.
 - Moved PWR LowPower callback to PLATFORM layers.
 - Bug fixes:
 - * Fixed wrong compensation of SysTicks during tickless idle.

- * Reinit RTC bus clock after exit from PM3 (power down).
- [OTA]
 - Initial support for OTA using the external flash.
- [platform]
 - Implemented platform specific time stamp APIs over OSTIMER.
 - Implemented platform specific APIs for OTA and external flash support.
 - Removed PLATFORM_GetLowpowerMode API.
 - Added support of CPU2 wake up over Spinel for OpenThread stack.
 - Bug fixes:
 - * Fixed issues related to handling CPU2 power state.
- [board]
 - Updated flash_config to support 64MB range.
- [linker script]
 - Fixed wrong assert.

6.1.1: KW45/K32W1 MR1

- [platform] Use new FLib_MemSet32Aligned() to write in ECC RAM bank to force ECC calculation in the MEM_ReinitRamBank() function
- [FunctionLib] Implement new API to set a word aligned
- [platform] Set coarse amplifier gain of the oscillator 32k to 3
- [platform] Switch back to RNG for MAC Address generation
- [SecLib] Get rid of the lowpower constraint of deep sleep in ECDH API
- [DCDC] Set DCDC output voltage to 1.35V in case LDO core is set to 1.1V to ensure a drop of 250mV between them
- [NVM] NvIdle() is now returning the number of operations that has been executed
- [documentation] Add markdown of each framework module by default on all package
- [LowPower] Add a delay advised by hardware team on exit of lowpower for SPC
- [SecLib] Rework of SecLib_mbedtls ECDH functions
- [OTA] Make OTA_IsTransactionPending() public API
- [FunctionLib] Change prototype of FLib_MemCpyWord(), pDst is now a void* to permit more flexibility
- [NVM] Add an API to know if there is a pending operation in the queue
- [FSCI] Fix wrong error case handling in FSCI_Monitor()

6.1.0: KW45/K32W1 RFP

- [LowPower] Do not call PLATFORM_StopWakeUpTimer() in PWR_EnterLowPower() if PLATFORM_StartWakeUpTimer() was not previously called
- [boards] Add the possibility to wakeup on UART 0 even if it is not the default UART
- [boards] Add support for Hardware flow control for UART0, Enable with gBoard-UseUart0HwFlowControl, Pin mux update with two additional API for RTS, CTS pins

- [Sensors] Improve ADC wakeup time from deep sleep state: use save and restore API for ADC context before/after deep sleep state.
- [linker script] update SMU2 shared memory region layout with NBU: increase sqram_btblebuf_size to support 24 connections. Shared memory region moved to the end
- [SecLib] SecLib_DeriveBluetoothSKD() API update to support if EdgeLock key shall be re-generated

6.0.11: KW45/K32W1 PRC3.1

FSCI: Framework Serial Communication Interface

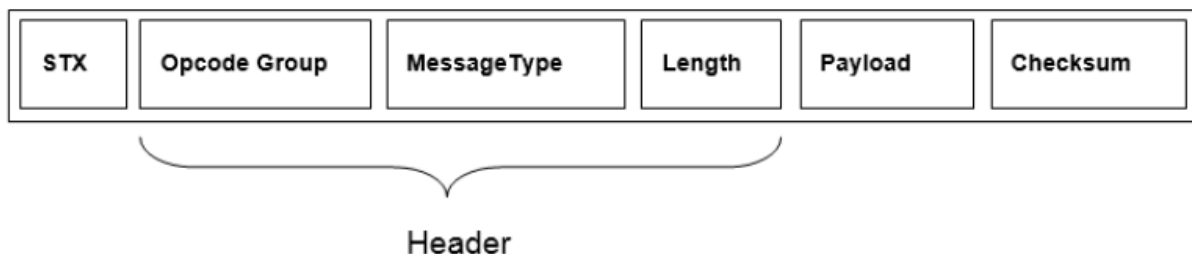
Overview The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup, the host processor (typically running the upper layers of a protocol stack) and the Black Box application (typically containing the lower layers of the stack, serving as a modem). The Test Tool software is an example of a host processor, which can interact with FSCI Black Boxes at various layers. In this setup, the user can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device enables monitoring of specific interfaces and API calls. Additionally, the FSCI injects or calls specific events and commands into the interfaces between layers.

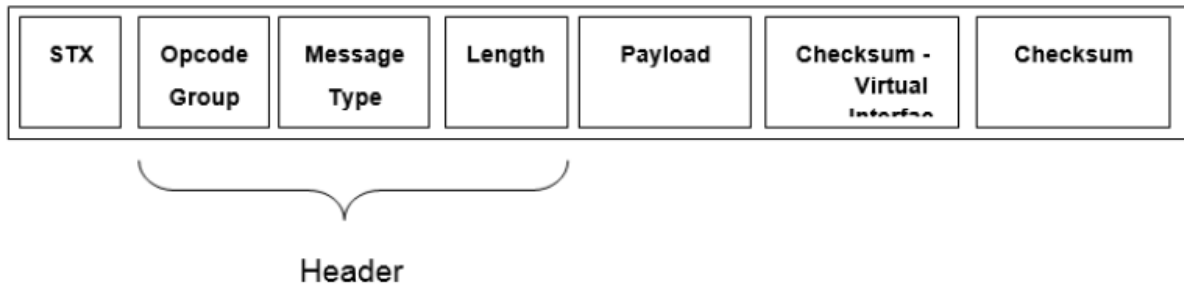
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but they can do so on different interfaces. For example, two MAC instances can register the same opcodes, one over UARTA, and the other over UARTB. This way, Test Tool can communicate with each MAC layer over two UART interfaces.

The FSCI module executes either in the context of the Serial Manager task or owns its dedicated task if the compilation Macro *gFsciUseDedicatedTask_c* is set to 1.

FSCI packet structure The FSCI module sends and receives messages as shown in the figure below. This structure is not specific to a serial interface and is designed to offer the best communication reliability. The Black Box device expects messages in little-endian format. It also responds with messages in little-endian format.



Below is an illustration of the FSCI packet structure when a virtual interface is used instead :



Field name	Length (bytes)	Description
STX	1	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1	Distinguishes between different Service Access Primitives (for example MLME or MCPS).
Message Type	1	Specifies the exact message opcode that is contained in the packet.
Length	1 or 2	The length of the packet payload, excluding the header and FCS. The length field content must be provided in little-endian format.
Payload	variable	Payload of the actual message.
Checksum	1	Checksum field used to check the data integrity of the packet.
Checksum2	0 or 1	The second CRC field appears only for virtual interfaces.

NOTE : When virtual interfaces are used, the first checksum is decremented with the ID of the interface. The second checksum is used for error detection.

constant definition The following Macro configures the FSCI module

```

#define gFsciIncluded_c 0 /* Enable/Disable FSCI module */
#define gFsciUseDedicatedTask_c 1 /* Enable Fsci task to avoid recursivity in Fsci module (Misra-compliant) */
#define gFsciMaxOpGroups_c 8
#define gFsciMaxInterfaces_c 1
#define gFsciMaxVirtualInterfaces_c 0
#define gFsciMaxPayloadLen_c 245 /* bytes */
#define gFsciTimestampSize_c 0 /* bytes */
#define gFsciLenHas2Bytes_c 0 /* boolean */
#define gFsciUseEscapeSeq_c 0 /* boolean */
#define gFsciUseFmtLog_c 0 /* boolean */
#define gFsciUseFileDataLog_c 0 /* boolean */
#define gFsciLoggingInterface_c 1 /* [0..gFsciMaxInterfaces_c) */
#define gFsciHostMacSupport_c 0 /* Host support at MAC layer */

```

The following provides the OpGroups values reserved by MAC, application, and FSCI.

FSCI Host FSCI Host is a functionality that allows separation at a certain stack layer between two entities, usually two boards running separate layers of a stack.

Support is provided for functionality at the MAC layer, for example, MAC/PHY layers of a stack are running as a Black Box on a board, and MAC higher layers are running on another. The higher layers send and receive serial commands to and from the MAC Black Box using the FSCI set of operation codes and groups.

The protocol of communication between the two is the same. The current level of support is provided for:

- FSCI_MsgResetCPUReqFunc – sends a CPU reset request to black box
- FSCI_MsgWriteExtendedAdrReqFunc – configures MAC extended address to the Black Box
- FSCI_MsgReadExtendedAdrReqFunc – N/A

The approach on the Host interfacing a Black Box using synchronous primitives is by default the polling of the FSCI_receivePacket function, until the response is received from the Black Box. The calling task polls whenever the task is being scheduled. This is required because a stack synchronous primitive requires that the response of that request is available in the context of the caller right after the SAP call has been executed.

The other option, available for RTOS environments, is using an event mechanism. The calling task blocks waiting for the event that is sent from the Serial Manager task when the response is available from the Black Box. This option is disabled by default. The disadvantage of this option is that the primitive cannot be received from another Black Box through a serial interface because the blocked task is the Serial Manager task, which reaches a deadlock as cannot be released again.

FSCI ACK ACK transmission is enabled through the gFsciTxAck_c macro definition. Each FSCI valid packet received triggers an FSCI ACK packet transmission on the same FSCI interface that the packet was received on. The serial write call is performed synchronously to send the ACK packet before any other FSCI packet. Only then the registered handler is called to process the received packet. The ACK is represented by the gFSCI_CnfOpcodeGroup_c and mFsciMsgAck_c Opcode. An additional byte is left empty in the payload so that it can be used optionally as a packet identifier to correlate packets and ACKs. ACK reception is the other component that is enabled through gFsciRxAck_c. The behavior is such that every FSCI packet sent through a serial interface triggers an FSCI ACK packet reception on the same interface after the packet is sent. If an ACK packet is received, the transmission is considered successful. Otherwise, the packet is re-sent a number of times. The ACK wait period is configurable through mFsciRxAckTimeoutMs_c and the number of transmission retries through mFsciTxRetryCnt_c. The ACK mechanism described above can also be coupled with a FSCI packet reception timeout enabled through gFsciRxTimeout_c and configurable through mFsciRxRestartTimeoutMs_c. Whenever there are no more bytes to be read from a serial interface, a timeout is configured at the predefined value if no other bytes are received. If new bytes are received, the timer is stopped and eventually canceled at successful reception. However, if, for any reason, the timeout is triggered, the FSCI module considers that the current packet is invalid, drops it, and searches for a new start marker.

FSCI usage example Detailed data types and APIs are described in ConnFWK API documentation.

Initialization

```
/* Configure the number of interfaces and virtual interfaces used */
#define gFsciMaxInterfaces_c 4
#define gFsciMaxVirtualInterfaces_c 2
....
/* Define the interfaces used */
static const gFsciSerialConfig_t myFsciSerials[] = {
    /* Baudrate, interface type, channel No, virtual interface */ {gUARTBaudRate115200_c, gSerialMgrUart_
    ↪ c, 1, 0}, {gUARTBaudRate115200_c, gSerialMgrUart_c, 1, 1}, {0 , gSerialMgrIICSlave_c, 1, 0}, {0 ,
    ↪ gSerialMgrUSB_c, 0, 0},
};
....
/* Call init function to open all interfaces */
FSCI_Init( (void*)mFsciSerials );
```

Registering operation groups

```
myOpGroup = 0x12; // Operation Group used
myParam = NULL; // pointer to a parameter to be passed to the handler function (myHandlerFunc)
myInterface = 1; // index of entry from myFsciSerials
...
FSCI_RegisterOpGroup( myOpGroup, gFsciMonitorMode_c, myHandlerFunc, myParam, myInterface );
```

Implementing handler function

```
void fsciMcpsReqHandler(void *pData, void* param, uint32_t interfaceId)
{
    clientPacket_t *pClientPacket = ((clientPacket_t*)pData);
    fsciLen_t myNewLen;
    switch( pClientPacket->structured.header.opCode )
    {
        case 0x01:
        {
            /* Reuse packet received over the serial interface The OpCode remains the same. The length of the
            ↪ response must be <= that the length of the received packet */
            pClientPacket->structured.header.opGroup = myResponseOpGroup; /* Process packet */
            ...
            pClientPacket->structured.header.len = myNewLen;
            FSCI_transmitFormattedPacket(pClientPacket, interfaceId);
            return;
        }
        case 0x02:
        {
            /* Allocate a new message for the response. The received packet is Freed */
            clientPacket_t *pResponsePkt = MEM_BufferAlloc( sizeof(clientPacketHdr_t) + myPayloadSize_d_
            ↪ + sizeof(uint8_t) // CRC);
            if(pResponsePkt)
            {
                /* Process received data and fill the response packet */ ...
                pResponsePkt->structured.header.len = myPayloadSize_d;
                FSCI_transmitFormattedPacket(pClientPacket, interfaceId);
            }
            break;
        }
        default:
            MEM_BufferFree( pData );
            FSCI_Error( gFsciUnknownOpcode_c, interfaceId );
            return;
    }
    /* Free message received over the serial interface */
    MEM_BufferFree( pData );
}
```

Helper Functions Library

Overview This framework provides a collection of features commonly used in embedded software centered on memory manipulation.

HWParameter: Hardware parameter

Production Data Storage Hardware parameters provide production data storage

Overview Different platforms/boards need board/network node-specific settings to function according to the design. (Examples of such settings are IEEE® addresses and radio calibration values specific to the node.) For this purpose, the last flash sector is reserved and contains hardware-specific parameters for production data storage. These parameters pertain to the network node as a distinct entity. For example, a silicon mounted on a PCB in a specific configuration, rather than to just the silicon itself. This sector is reserved by the linker file, through the PROD_DATA section and it should be read/written only through the API described below.

Note : This sector is not erased/written at code download time and it is not updated via over-the-air firmware update procedures to preserve the respective node-specific data, regardless of the firmware running on it.

Constant Definitions Name :

```
extern uint32_t PROD_DATA_BASE_ADDR[];
```

Description :

This symbol is defined in the linker script. It specifies the start address of the PROD_DATA section.

Name :

```
static const uint8_t mProdDataIdentifier[10] = {"PROD_DATA:"};
```

Description :

The value of this constant is copied as identification word (header) at the beginning of the PROD_DATA area and verified by the dedicated read function.

Note: the length of mProdDataIdentifier imposes the definition of PROD_DATA_ID_STRING_SZ as 10. The legacy HW parameters structure provides headroom for future usage. There are currently 63 bytes available.

Data type definitions Name :

```
typedef PACKED_STRUCT HwParameters_tag
{
    uint8_t identificationWord[PROD_DATA_ID_STRING_SZ]; /* internal usage only: valid data present */
    /*@{*/
    uint8_t bluetooth_address[BLE_MAC_ADDR_SZ]; /*!< Bluetooth address */
    uint8_t ieee_802_15_4_address[IEEE_802_15_4_SZ]; /*!< IEEE 802.15.4 MAC address - K32W1 only
    /*
    uint8_t xtalTrim; /*!< XTAL 32MHz Trim value */
    uint8_t xtalCap32K; /*!< XTAL 32kHz capacitance value */
    /* For forward compatibility additional fields may be added here
    Existing data in flash will not be compatible after modifying the hardwareParameters_t typedef.
    In this case the size of the padding has to be adjusted.
    */
    uint8_t reserved[1];
    /* first byte of padding : actual size if 63 for legacy HwParameters but
    complement to 128 bytes in the new structure */
}
hardwareParameters_t;
```

Description:

Defines the structure of the hardware-dependent information.

Note : Some members of this structure may be ignored on a specific board/silicon configuration. Also, new members may be added for implementation-specific purposes and the backward compatibility must be maintained.

The CRC calculation starts from the reserved field of the hardwareParameters_t and ends before the hardwareParamsCrc field. Additional members to this structure may be added using the following method :

Add new fields before the reserved field. This method does not cause a CRC fail, but you must keep in mind to subtract the total size of the new fields from the size of the reserved field. For example, if a field of uint8_t size is added using this method, the size of the reserved field shall be changed to 63.

Co-locating application factory data in HW Parameters flash sector. The sector containing the Hardware parameter structure may be located in the internal flash, usually at its last sector. The actual Hardware parameter structure has a size of 128 bytes - including padding reserved for future use. Since there is plenty of room available in a flash sector (4kB or 8kB), co-locating Application Factory Data in the same structure prevents from reserving another flash sector for these data. The application designer may adopt this solution by defining gHwParamsAppFactoryDataExtension_d as 1. A total of 2kB is allotted to this purpose.

If this option was chosen, whenever any of the Hardware parameter fields is modified, its CRC16 will change so the sector will need erasing. The gHwParamsAppFactoryDataPreserveOnHwParamUpdate_d compilation option deals with restoring the contents of the App Factory Data. Nonetheless this requires a temporary allocation a 2kB buffer to preserve the previous content and restore then on completion of the Hw Parameter update.

Special reserved area at start of IFR1 in range [0x02002000..0x02002600] On development boards a 1536 byte area is reserved and the actual Hardware parameter area begins at offset 0x600. Preserving this area on a HW parameter update also requires a temporary 1.5kB dynamic allocation (in addition to the App Factory 2kB allocation), to be able to restore on completion of update operation.

HW Parameters Production Data placement options The placement of production data (PROD_DATA) can be selected based on the definition of gHwParamsProdDataPlacement_c (see fwk_config.h). The productions data seldom need update for final products, once calibration data, MAC addresses or others have been programmed. Two cases exist, plus a transition mode :

1) gHwParamsProdDataMainFlashMode_c (0) :

- PROD_DATA are located at top of Main Flash. Hardware parameters section is placed in the last sector of internal flash [0xfe000..0x100000[.
- The linker script must reserve this area explicitly so as to prevent placement of NVM or text sections at that location by setting gUseProdInfoMainFlash_d.

2) gHwParamsProdDataMainFlash2IfirMode_c(1) : - PROD_DATA are located in IFR1, but Main-Flash version still exists during interim period. - If the contents of the PROD_DATA section in MainFlash is valid (not blank and correct CRC) but the IFR PROD_DATA is still blank, copy the contents of MainFlash PROD_DATA to IFR location. - When done PROD_DATA in IFR are used. Once the transition is done, an application using (2: gHwParamsProdDataPlacementIfirMode_c) may be programmed.

3) gHwParamsProdDataIfirMode_c (2) :

- PROD_DATA section dwells in the IFR1 sector [0x02002000..0x02004000[
- in development phase the area comprised between [0x02002000..0x02002600[must be reserved for internal purposes.
- This allows to free up the top sector of Main Flash by linking with gUseProdInfoMainFlash_d unset.

LowPower

Low Power reference user guide This Readme file describes the connectivity software architecture and provides the general low power enablement user guide.

1- Connectivity Low Power SW architecture The connectivity low power software architecture is composed of various components. These are described from the lower layer to the application layer:

1. The SDK power manager in component/power_manager. This component provides the basic low power framework. It is not specific to the connectivity but generic across devices. it covers:
 - gather the low power constraints for upper layer and take the decision on the best suitable low power state the device is allowed to go to fullfill the constraints.
 - call the low power entry and exit function callbacks
 - call the appropriate SW routines to switch the device into the suitable low power state
2. Connectivity Low power module in the connectivity framework. This module is composed of:
 - The low power service called PWR inside framework/LowPower (this folder), This module is generic to all connectivity devices.
 - The platform lowpower: fwk_platform_lowpower.[ch] located in framework\platform\<platform_name>. These files are a collection of low power routines functions for the PWR module and upper layer. These are specific to the device.Both PWR and platform lowpower files are detailed in section below.
3. Low power Application modules, it consists of 3 parts:
 - Application initialization file app_services_init.c where the application initializes the low power framework, see next section 'Demo example for typical usage of low power framework'
 - Application Idle task from application to call the main low power entry function PWR_EnterLowPower() to switch the device into lowpower. This function is application specific, one example is given in the section 1.3.3
 - Low power board files : board_lp.[ch] located in board/lowpower. These files implement the low power entry and exit functions related to the application and board. Customers shall modify these files for their own needs. Example code is given for the connectivity applications.

User guide is provided in section 1.3 below.

Note : Linker script may also be impacted for power down mode support in order to provide an RAM area for ROM warm boot (depends on the platform) and application warmboot stack

The Low power central and master reference design applications provide an example of Low power implementation for BLE. Customer can also refer to the associated document 'low power connectivity reference design user guide'.

1.1 - SDK power manager This module provides the main low power functionalities such as:

- Decide the best low-power mode dependent on the constraints set by upper layers by using PWR_SetLowPowerModeConstraints() API function.

- Handle the sequences to enter and exit low-power mode.
- Enable and configure wake up sources, call the application callbacks on low power entry/exit sequences.

The SDK power manager provides the capability for application and all components to receive low power constraints to the power. The Application does not set the low-power mode the device shall go into. When going to low power, the SDK power manager selects the best low-power mode that fits all the constraints.

As an example, if the low power constraint set from Application is Power Down mode, and no other constraint is set, the SDK power manager selects Power down mode, the next time the device enters low power. However, if a new constraint is set by another component, such as the SecLib module that operates Hardware encryption, the SecLib module would select WFI as additional low power constraint. Also, the SDK power manager selects this last low-power mode until the constraint is released by the SecLib module. It then reselects Power Down mode for further low power entry modes.

1.2 - PWR Low power module The PWR module in the connectivity framework provides additional services for the connectivity stacks and applications on top of the SDK power manager.

It also provides a simple API for Connectivity Stack and Connectivity applications.

However, more advanced features such as configuring the wake-up sources are only accessible from the SDK Power Manager API.

In addition to the SDK Power Manager, the PWR module uses the software resources from lower level drivers but is independent of the platform used.

1.2.1 - Functional description Initialization of the PWR module should be done through PWR_Init() function. This is mainly to initialize the SDK power manager and the platform for low power. It also registers PWR low power entry/exit callback PWR_LowpowerCb() to the SDK power manager. This function will be called back when entering and exiting low power to perform mandatory save/restore operations for connectivity stacks. The application can perform extra optional save/restore operations in the board_lp file where it can register to the SDK Power Manager its own callback. This is usually used to handle optional peripherals such as serial interfaces, GPIOs, and so on. The main entry function is PWR_EnterLowPower(). It should be called from Idle task when no SW activity is required. The maximum duration for lowpower is given as argument timeoutUs in useconds. This function will check the next Hardware event in the connectivity stack, typically the next Radio activity. A wakeup timer is programmed if the timeoutUs value is shorter than the next radio event timing. Passing a timeout of 0us will be interpreted as no timeout on the application side.

On device wakeup from low power state, the function will return the time duration the device has been in low power state.

Two API are provided to set and release low power state constraints : PWR_SetLowPowerModeConstraint() and PWR_ReleaseLowPowerModeConstraint(). These are helper functions. User can use directly the SDK power manager if needed.

The PWR module also provides some API to be set as callbacks into other components to prevent from going to low power state. It can be used in following examples :

1. If a DMA is running, the module in charge of the DMA would need to set a constraint to avoid the system from going to a low power state when the RAM and system bus are no longer available.
2. If transfer is going on a peripheral, the drivers shall set a constraint to forbid low power mode.
3. If encryption is on going through an Hardware accelerator, the HW accelerator and the required resources (clocks, etc), shall be kept active also by setting a constraints.

1.2.2 - Tickless mode support This module also provides some routines functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() from PWR_systicks.c in order to support the tickless mode when using FreeRTOS. The tickless mode is the capability to suspend the periodic system ticks from FreeRTOS and keep timebase tracking using another low power counter. In this implementation, the Timer Manager and time_stamp component are used for this purpose.

Idle task shall call these functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() before and after the call to the main low power entry function PWR_EnterLowPower().

Refer to framework/LowPower/PWR_systicks.c file or section 2.1 below for more information.

1.3 - Low power platform submodule Low power platform module file fwk_platform_lowpower.c provides the necessary helper functions to support low power device initialization, device entry, and exit routines. These are platform and device specific. Typically, the PWR module uses the low power platform submodule for all low power specific routines.

The low power platform submodule is documented in the Connectivity Framework Reference Manual document and in the Connectivity Framework API document.

1.4 - Low power board files Low power board files board_lp.[ch] are both application and board specific. Users should update this file to add new functions to include new used peripherals that require low power support. In the current SDK package, only Serial Manager over UART and button (IO toggle wake up source) are supported and demonstrated in the Bluetooth LE demo application.

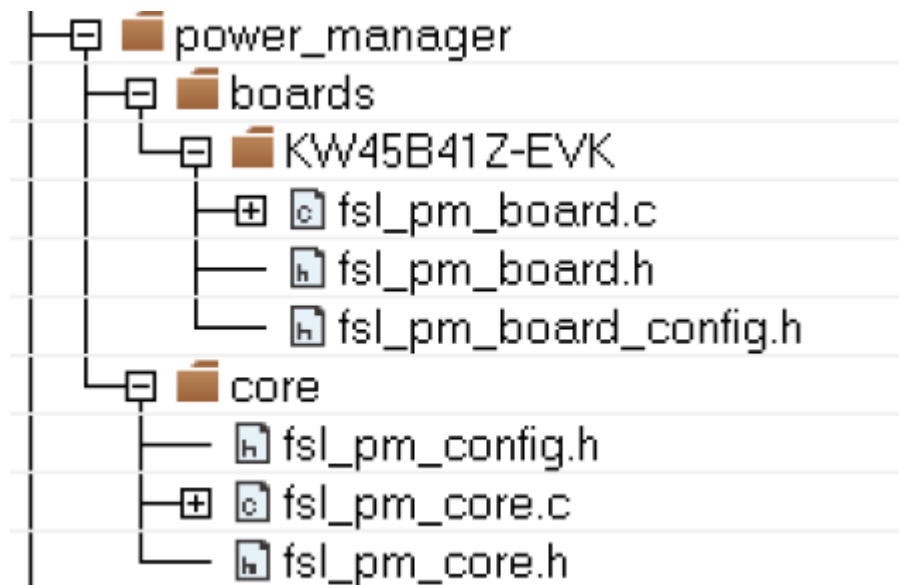
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

2 - Low power Application user guide This section provides a user guide to enable Low power on a connectivity application, It gives example of typical implementation for the initialization, Idle task function and low power entry/exit functions.

2.1 - Application Project updates It is recommended to reuse the low-power peripheral/central reference design application projects as a start. This ensures that everything is in place for the low-power optimization feature. Then, application files may be added to one of the two projects.

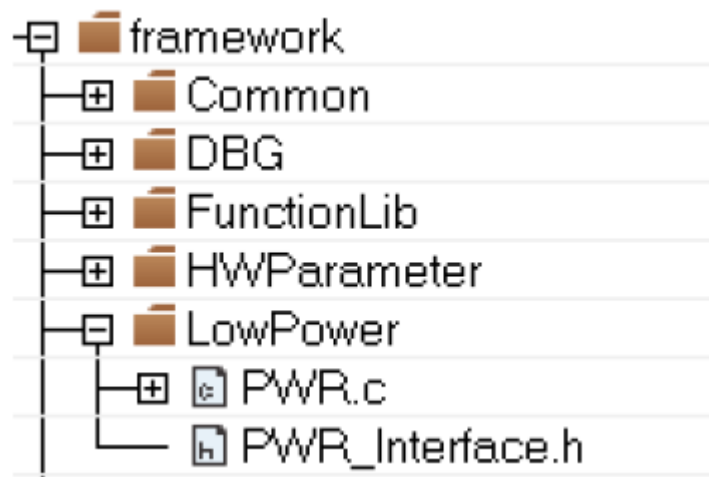
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

2.1.1 - SDK Power Manager Most of the Low power functionality is implemented in the SDK Power Manager. The files to add into the project SDK power_manager module are listed in the figure below:



You need to use the files located in the folder that match your device.

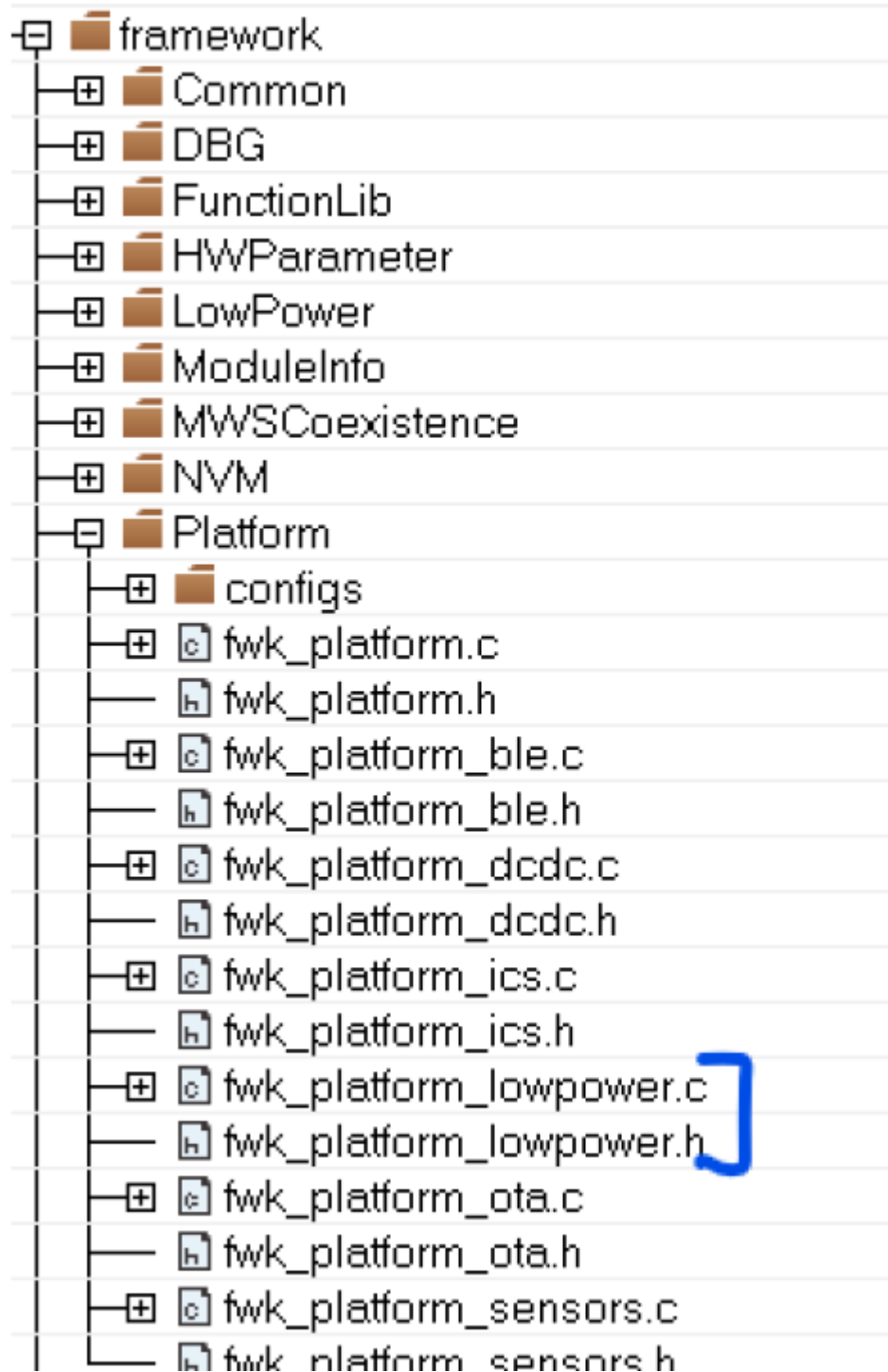
2.1.2 - PWR connectivity framework module PWR.c PWR_Interface.h shall be added to your application projects :



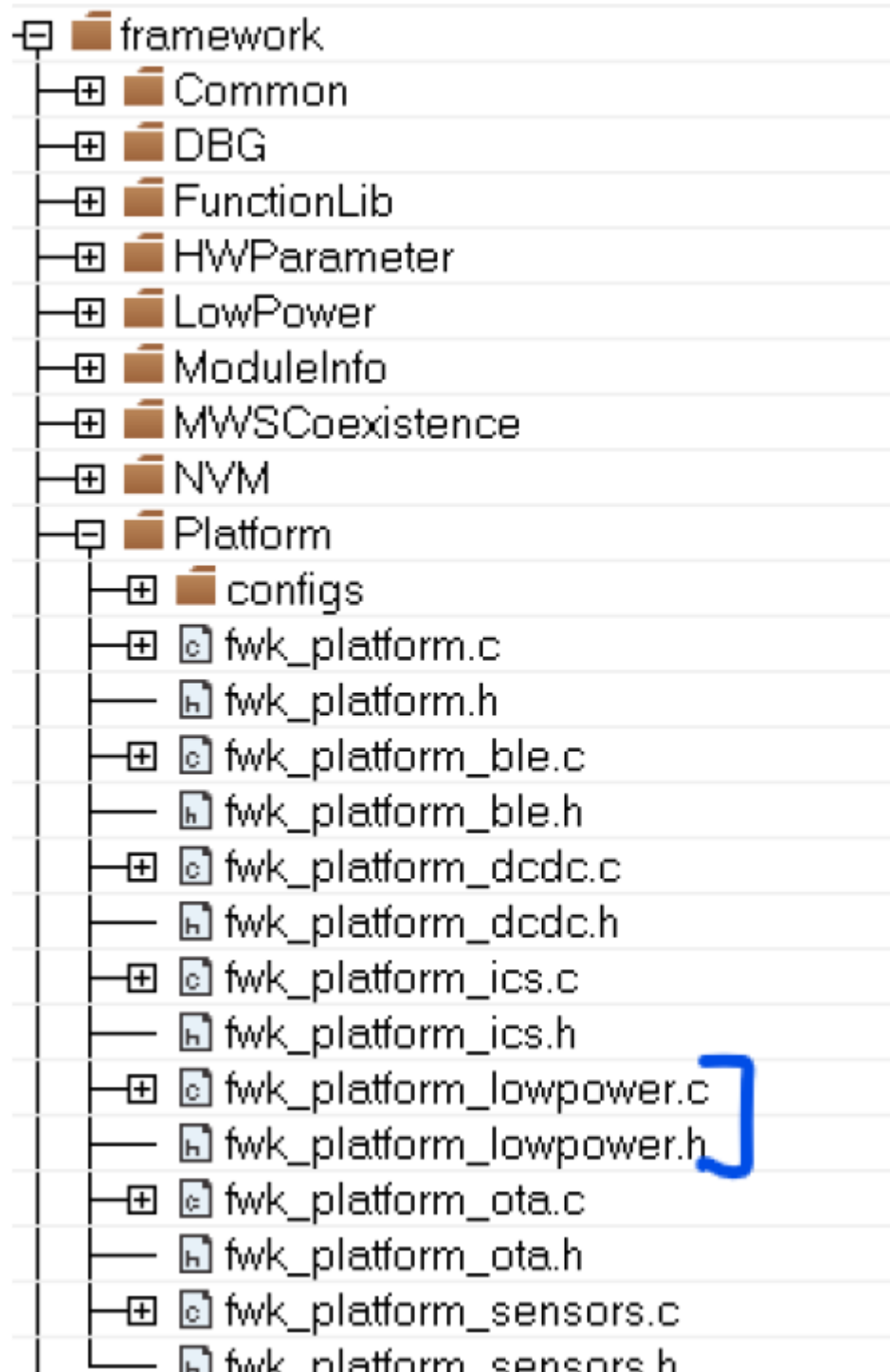
Optionally, in order to support SysTick less mode, `PWR_systicks.c` or `PWR_systicks_bm.c` could also be added.

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

2.1.3 -Low power platform submodule Low power platform files can be found in the 'Platform' module in the connectivity framework:



2.1.4 - Low power board files These files are located in the same folder that the other board files `board.[ch]`. Hence, it is not required to add any new include path at compiler command line.



2.1.5 - Application RTOS Idle hook and tickless hook functions See section 2.4.3 Idle task implementation example

2.2 - Low power and wake up sources Initialization Low power initialization and configuration are performed in APP_ServiceInitLowpower() function. This is called from APP_InitServices() function called from the main() function so all is already set up when calling the main application entry point, typically BluetoothLEHost_AppInit() function in the Bluetooth LE demo applications.

The default Low Power mode configured in APP_InitServices() is Deep Sleep mode. In Bluetooth

LE, (or any other stack technology), Deep Sleep mode fits for all use cases. For instance, for Bluetooth LE states: Advertising, Connected, Scanning states. This mode already performs a very good level of power saving and likely, this is not required to optimize more if the device is powered from external supply.

APP_ServiceInitLowpower() function performs the following initialization and configuration:

- Initialize the Connectivity framework Low power module PWR_Init(), this function initialized the SDK power manager.
- Configure the wakeup sources such as serial manager wake up source for UART, or button for IO wake up configuration. These are typical wakeup sources used in the connectivity application. Developer may want to add additional wake up sources here specific for the application.

Note : The low power timer wakeup source and wakeup from Radio domain are directly enabled from the Connectivity framework Low power module PWR as it is mandatory for the connectivity stack. If your application supports other peripherals (such as i2c, spi, and others) that require wake sources from low power, developer should add additional wake up sources setting in this function APP_ServiceInitLowpower(). The complete list of wakeup sources are available from the SDK power manager component, see file fsl_pm_board.h in component/boards/<device_name>/.

- Initialize and register the Low power board file used to register and implement low power entry and exit callback function used for peripheral. This is done by calling the BOARD_LowPowerInit() function.
- Register low power Enter and exit critical function to driver component to enable / disable low power when the Hardware is active. Example is given for serial manager that needs to disable low power when the TX ring buffer contains data so the device does not enter low power until the buffer is empty.

Finally, APP_ServiceInitLowpower() function configures the Deep Sleep mode as the default low power constraint for the application. It is recommended to keep this level of low power constraint during all the connectivity stack initialization.

Example of low power framework initialization can be found in app_services_init.c file. Below is some code example for initializing the low power framework and wake up sources:

```
static void APP_ServiceInitLowpower(void)
{
    PWR_ReturnStatus_t status = PWR_Success;

    /* It is required to initialize PWR module so the application
     * can call PWR API during its init (wake up sources...) */
    PWR_Init();

    /* Initialize board_lp module, likely to register the enter/exit
     * low power callback to Power Manager */
    BOARD_LowPowerInit();

    /* Set Deep Sleep constraint by default (works for All application)
     * Application will be allowed to release the Deep Sleep constraint
     * and set a deepest lowpower mode constraint such as Power down if it needs
     * more optimization */
    status = PWR_SetLowPowerModeConstraint(PWR_DeepSleep);
    assert(status == PWR_Success);

    #if (defined(gAppButtonCnt_c) && (gAppButtonCnt_c > 0))

    /* Init and enable button0 as wake up source
     * BOARD_WAKEUP_SOURCE_BUTTON0 can be customized based on board configuration
```

(continues on next page)

(continued from previous page)

```

    * On EVK we use the SW2 mapped to GPIOD */
    PM_InitWakeupSource(&button0WakeupSource, BOARD_WAKEUP_SOURCE_BUTTON0, NULL,
↪true);
#endif

#if (gAppButtonCnt_c > 1)
/* Init and enable button1 as wake up source
 * BOARD_WAKEUP_SOURCE_BUTTON1 can be customized based on board configuration
 * On EVK we use the SW3 mapped to PTC6 */
    PM_InitWakeupSource(&button1WakeupSource, BOARD_WAKEUP_SOURCE_BUTTON1, NULL,
↪true);
#endif

#if (defined(gAppUseSerialManager_c) && (gAppUseSerialManager_c > 0))

#if defined(gAppLpuart0WakeupSourceEnable_d) && (gAppLpuart0WakeupSourceEnable_d > 0)
/* To be able to wake up from LPUART0, we need to keep the FRO6M running
 * also, we need to keep the WAKE domain is SLEEP.
 * We can't put the WAKE domain in DEEP SLEEP because the LPUART0 is not mapped
 * to the WUU as wake up source */
    (void)PM_SetConstraints(PM_LP_STATE_NO_CONSTRAINT, APP_LPUART0_WAKEUP_
↪CONSTRAINTS);
#endif

/* Register PWR functions into SerialManager module in order to disable device lowpower
   during SerialManager processing. Typically, allow only WFI instruction when
   uart data are processed by serial manager */
    SerialManager_SetLowpowerCriticalCb(&gSerMgr_LowpowerCriticalCBs);
#endif

#if defined(gAppUseSensors_d) && (gAppUseSensors_d > 0)
    Sensors_SetLowpowerCriticalCb(&app_LowpowerSensorsCriticalCBs);
#endif

    (void)status;
}

```

2.3 - low power entry/exit sequences : board files updates Board Files that handles low-power are board_lp.[ch] files.

Low power board files implement the low-power callbacks of the peripherals to be notified when entering or exiting Low Power mode. This module also registers these low-power callbacks to the SDK Power Manager component to get the notifications when the device is about to enter low-power or exit Low Power mode. The Low-power callbacks are registered from BOARD_LowPowerInit() function. This function is called from app_services_init.c file after PWR module initialization.

The low power callback functions can be categorized in two groups:

- **Entry Low power call back functions:** These are usually used to prepare the peripherals to enter low-power. For example, they can be used for flushing FIFOs, switching off some clocks, and reconfiguring pin mux to avoid leakage on pins. In case of Power Down mode, these functions could be used to save the Hardware peripheral context.
- **Exit Low power call back functions:** These are typically used to restore the peripherals to functionality. Therefore, they perform the reverse of what is done by the entry call-back functions: restoring the pin mux, re-enabling the clock, in case of Power Down mode, restoring the Hardware peripheral context, and so on.

Note that distinction can be done between clock gating mode (Deep Sleep mode), and power gated mode (Power down mode) when entering and exiting Low Power mode. The

BOARD_EnterLowPowerCb() and BOARD_ExitLowPowerCb() functions provide the code to call the various peripheral entry and exit functions to go and exit Deep Sleep mode: serial manager, button, debug console, and others.

However, the processing to save and restore the Hardware peripheral is implemented in different functions BOARD_EnterPowerDownCb() and BOARD_ExitPowerDownCb(). These two functions should be called when exiting power gated modes of the power domain. These two should implement specific code for such case (likely the complete reinitialization of each peripheral). In order to know the Low Power mode that the wake up domain, or main domain has been entered, the low-power platform API PLATFORM_GetLowpowerMode() can be called.

Note : BOARD_ExitPowerDownCb() is called before BOARD_ExitLowPowerCb() as it is generally required to restore the Hardware peripheral contexts before reconfiguring the pin mux to avoid any signal glitches on the pads

Also, It is important to know whether the location of the Hardware peripheral is in the main domain or wake up domain. The two power domains can go into different power modes with the limitation that the wakeup domain cannot go to a deepest Low Power mode than the main domain. Depending on the constraint set on SDK power manager, the wake up domain could remain in active while the main domain can go to deep sleep or power down modes. In this case, the peripherals in the wake up domain does not required to be restored, as explained in the section Power Down. Likely, only pin mux reconfiguration is required in this case.

example Low power entry and exit functions shall be registered to the SDK power manager so these functions will be called when the device will enter and exit low power mode. This is done by BOARD_LowPowerInit() typically called from application source code in app_services_init.c file

```
static pm_notify_element_t boardLpNotifyGroup = {
    .notifyCallback = BOARD_LowpowerCb,
    .data          = NULL,
};

void BOARD_LowPowerInit(void)
{
    status_t status;

    status = PM_RegisterNotify(kPM_NotifyGroup2, &boardLpNotifyGroup);
    assert(status == kStatus_Success);
    (void)status;
}
```

BOARD_LowpowerCb() callback function will handle both the entry and exit sequences. An argument is passed to the function to indicate the lowpower state the device enter/exit. Typical implementation is given below. Customer shall make sure to differentiate low power entry and exit, and the various low power states.

Typically, nothing is expected to be done if low power state is WFI or Sleep mode. These modes are some light low power states and the system can be woken up by interrupt trigger.

In Deep sleep mode, the clock tree and source clocks are off, the system needs to be woken up from an event from the WUU module.

In Power down mode, some peripherals are likely to be powered off, context save and restore may need to be done in these functions.

```
static status_t BOARD_LowpowerCb(pm_event_type_t eventType, uint8_t powerState, void *data)
{
    status_t ret = kStatus_Success;
    if (powerState < PLATFORM_DEEP_SLEEP_STATE)
    {
        /* Nothing to do when entering WFI or Sleep low power state
           NVIC fully functionnal to trigger upcoming interrupts */
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    else
    {
        if (eventType == kPM_EventEnteringSleep)
        {
            BOARD_EnterLowPowerCb();

            if (powerState >= PLATFORM_POWER_DOWN_STATE)
            {
                /* Power gated low power modes often require extra specific
                 * entry/exit low power procedures, those should be implemented
                 * in the following BOARD API */
                BOARD_EnterPowerDownCb();
            }
        }
        else
        {
            /* Check if Main power domain really went to Power down,
             * powerState variable is just an indication, Lowpower mode could have been skipped by an
            ↪immediate wakeup
             */
            PLATFORM_PowerDomainState_t main_pd_state = PLATFORM_NO_LOWPOWER;
            PLATFORM_status_t status;

            status = PLATFORM_GetLowpowerMode(PLATFORM_MainDomain, &main_pd_state);
            assert(status == PLATFORM_Successful);
            (void)status;

            if (main_pd_state == PLATFORM_POWER_DOWN_MODE)
            {
                /* Process wake up from power down mode on Main domain
                 * Note that Wake up domain has not been in power down mode */
                BOARD_ExitPowerDownCb();
            }

            BOARD_ExitLowPowerCb();
        }
    }
    return ret;
}

```

2.4 - Low power constraint updates and optimization Except for the board file update as seen in previous section, the application does not need any other changes for low-power support in Deep Sleep mode. It shall work as if no low-power is supported. However, If more aggressive power saving is required, this constraint can be changed in your application in order to further reduce the power consumption in Low Power mode.

2.4.1 - Changing the Default Application low power constraint after firmware initialization The Low power reference design applications (central or peripheral) provides demonstration on how to change the Application low power constraint. In the Application main entry point `BluetoothLEHost_AppInit()`, Deep Sleep mode is configured by default from `APP_ServiceInitLowpower()` function.

Note : It is recommended to keep Deep Sleep mode as default during all the stack initialization phase until `BluetoothLEHost_Initialized()` and `BleApp_StartInit()` functions are called. In case of Bonded device with privacy, it is recommended to wait for `gControllerPrivacyStateChanged_c` event to be called.

BleApp_LowpowerInit() function provides an example of code on how to release the default Deep sleep low-power constraint and set a new constraint such as Power down mode for the application. This deeper low-power mode is used when no Bluetooth LE activity is on going, and if no other higher Low-power constraint is set by another components or layer. For instance, if some serial transmission is on going by the serial manager, or if the SecLib module has on going activity on the HW crypto accelerator, the low-power mode could less deep.

```
static void BleApp_LowpowerInit(void)
{
    #if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
        PWR_ReturnStatus_t status;

        /*
         * Optionally, Allow now Deepest lowpower mode constraint given by gAPP_
        ↪LowPowerConstraintInNoBleActivity_c
         * rather than DeepSleep mode.
         * Deep Sleep mode constraint has been set in APP_InitServices(), this is fine
         * to keep this constraint for typical lowpower application but we want the
         * lowpower reference design application to be more aggressive in term of power saving.

         * To apply a lower lowpower mode than Deep Sleep mode, we need to
         * - 1) First, release the Deep sleep mode constraint previously set by default in app_services_init()
         * - 2) Apply new lowpower constraint when No BLE activity
         * In the various BLE states (advertising, scanning, connected mode), a new Lowpower
         * mode constraint will be applied depending of Application Compilation macro set in app_preinclude.
        ↪h :
         * gAppPowerDownInAdvertising, gAppPowerDownInConnected, gAppPowerDownInScanning
         */

        /* 1) Release the Deep sleep mode constraint previously set by default in app_services_init() */
        status = PWR_ReleaseLowPowerModeConstraint(PWR_DeepSleep);
        assert(status == PWR_Success);
        (void)status;

        /* 2) Apply new Lowpower mode constraint gAppLowPowerConstraintInNoBleActivity_c */
        * The BleAppStart() call above has already set up the new lowpower constraint
        * when Advertising request has been sent to controller */
        BleApp_SetLowPowerModeConstraint(gAppLowPowerConstraintInNoBleActivity_c);
    #endif
}
```

2.4.2 - Changing the Application lowest low power constraint during application execution

In the various application use cases, (in the various Bluetooth LE activity states, advertising, connected, scanning), some lower low-power constraint can be set, as Power down for advertising, Deep Sleep for connected, or Scanning. Customer can change the level of Low Power mode in the various use case mainly depending of the time duration the device is supposed to remain in low-power. The longer the time that the device remains in low power, the higher the benefit for a deeper Low Power mode such as Power down mode. However, please note that the wake up from power down mode takes significantly more time than deep sleep as ROM code is re executed and the hardware logic needs to be restored. Sections Deep Sleep and Power Down provide some guidance on when to use Deep Sleep mode or Power Down modes respectively.

In the low power reference design applications, four application compilations macros are defined to adjust the low-power mode into advertising, scanning, connected, or no Bluetooth LE activity. Other use cases can be added as desired. For instance, If application needs to run a DMA transfer, or if application needs to wakeup regularly to process data from external device, it may be useful to set WFI constraint (in case of DMA transfer), or Deep Sleep constraint (in case of regular wake up to process external data), rather than power down or a even lower low-power mode.

The 4 application compilation macros can be found in app_preinclude.h file of the project. See

app_preinclude.h for low power reference design peripheral application :

```

/*! Lowpower Constraint setting for various BLE states (Advertising, Scanning, connected mode)
The value shall map with the type definition PWR_LowpowerMode_t in PWR_Interface.h
0 : no LowPower, WFI only
1 : Reserved
2 : Deep Sleep
3 : Power Down
4 : Deep Power Down
Note that if a Ble State is configured to Power Down mode, please make sure
gLowpowerPowerDownEnable_d variable is set to 1 in Linker Script
The PowerDown mode will allow lowest power consumption but the wakeup time is longer
and the first 16K in SRAM is reserved to ROM code (this section will be corrupted on
each power down wakeup so only temporary data could be stored there.)
Power down feature not supported. */

#define gAppLowPowerConstraintInAdvertising_c      3
/* Scanning not supported on peripheral */
// #define gAppLowPowerConstraintInScanning_c      2
#define gAppLowPowerConstraintInConnected_c      2
#define gAppLowPowerConstraintInNoBleActivity_c   4

```

In `lowpower_central.c` `lowpower_preripheral.c` files, the application sets and releases the low power constraint from `BleApp_SetLowPowerModeConstraint()` and `BleApp_ReleaseLowPowerModeConstraint()` functions. These functions are called with the macro value passed as argument.

Important Note : Setting the application low power constraint shall be done on new Bluetooth LE state request so the new constraint is applied immediately, while the application low-power mode constraint shall be released when the Bluetooth LE state is exited. For example, setting the new low power constraint for Advertising shall be done when the application requests advertising to start. Releasing the low power constraint shall be done in the advertising stop callback (advertising has been stopped).

After releasing the low power constraint, the previous low power constraint, (likely the one that has been set during firmware initialization in `APP_ServiceInitLowpower()` function, or the updated low power constraint in `BleApp_StartInit()` function) applies again.

2.4.3 - Idle task implementation example

2.4.3.1 Tickless mode support and Low power entry function Idle task configuration from FreeRTOS shall be enabled by `configUSE_TICKLESS_IDLE` in `FreeRTOSConfig.h`. This will have the effect to have `vPortSuppressTicksAndSleep()` called from Idle task created by FreeRTOS. Here is a typical implementation of this function:

```

void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{
    bool abortIdle = false;
    uint64_t actualIdleTimeUs, expectedIdleTimeUs;

    /* The OSA_InterruptDisable() API will prevent us to wakeup so we use
    * OSA_DisableIRQGlobal() */
    OSA_DisableIRQGlobal();

    /* Disable and prepare systicks for low power */
    abortIdle = PWR_SysticksPreProcess((uint32_t)xExpectedIdleTime, &expectedIdleTimeUs);

    if (abortIdle == false)
    {

```

(continues on next page)

(continued from previous page)

```

/* Enter low power with a maximal timeout */
actualIdleTimeUs = PWR_EnterLowPower(expectedIdleTimeUs);

/* Re enable systicks and compensate systick timebase */
PWR_SysticksPostProcess(expectedIdleTimeUs, actualIdleTimeUs);
}

/* Exit from critical section */
OSA_EnableIRQGlobal();
}

```

2.4.3.2 Connectivity background tasks and Idle hook function example Some process needs to be run in background before going into low power. This is the case for writing in NVM, or firmware update OTA to be written in Flash. If so, configUSE_IDLE_HOOK shall be enabled in FreeRTOSConfig.h so vApplicationIdleHook() will be called prior to vPortSuppressTicksAndSleep(). Typical implementation of vApplicationIdleHook() function can be found here :

```

void vApplicationIdleHook(void)
{
    /* call some background tasks required by connectivity */
    #if ((gAppUseNvm_d) || \
        (defined gAppOtaASyncFlashTransactions_c && (gAppOtaASyncFlashTransactions_c > 0)))

        if (PLATFORM_CheckNextBleConnectivityActivity() == true)
        {
            BluetoothLEHost_ProcessIdleTask();
        }
    #endif
}

```

PLATFORM_CheckNextBleConnectivityActivity() function implemented in low power platform file fwk_platform_lowpower.c typically checks the next connectivity event and returns true if there's enough time to perform time consuming tasks such as flash erase/write operations (can be defined by the compile macro depending on the platform).

2. Low power features

2.1 - FreeRTOS systicks Low power module in framework supports the systick generation for FreeRTOS. Systicks in FreeRTOS are most of the time not required in the Bluetooth LE demos applications because the framework already supports timers by the timer manager component, so the application can use the timers from this module. The systicks in FreeRTOS are useful for all internal timer service provided by FreeRTOS (through OSA) like OSA_TimeDelay(), OSA_TimeGetMsec(), OSA_EventWait(). When systicks are enabled, an interrupt (systick interrupt) is triggered and executed on a periodic basis. In order to save power, periodic systick interrupts are undesirable and thus disabled when going to low-power mode. This feature is called low power FreeRTOS tickless mode. When entering the low power state, the system ticks shall be disabled and switch to a low power timer. On wake-up, the module retrieves the time passed in low power and compensate the ticks count accordingly. This feature does not apply on bare metal scheduler.

On FreeRTOS, the vPortSuppressTicksAndSleep() function implemented in the app_low_power.c file will be called when going to idle. FreeRTOS will give to this function the xExpectedIdleTime, time in tick periods before a task is due to be moved into the Ready state. This function will manage the systicks (disable/enable) through PWR_SysticksPreProcess() and PWR_SysticksPostProcess() calls. Then, when calling PWR_EnterLowPower(), a time out duration in micro seconds will be given and the function will set a timer before entering low power.

In addition, this function will return the low power period duration, used to compensate the ticks count.

In our example low power reference design peripheral application, an `OSA_EventWait()` has been added to demonstrate the tickless mode feature. You can adjust the timeout with the `gApp-TaskWaitTimeout_ms_c` flag in the `app_preinclude.h` file, its value in our demo is 8000ms. So 8 seconds after stopping any activity we will wake up from low power. If the flag is not defined in the application its value will be `osaWaitForever_c` and there will be no OS wake up.

2.2 - Selective RAM bank retention To optimize the consumption in low power, the linker script specific function `PLATFORM_GetDefaultRamBanksRetained()` is implemented. This function obtains the RAM banks that need to be retained when the device goes in low power, in order to set them with `PLATFORM_SetRamBanksRetained()` function. The RAM banks that are not needed are set in power off state, when the device goes in low power mode.

The function `PLATFORM_GetDefaultRamBanksRetained()` is linker script specific. Hence, it cannot be adapted for a different application. If these functions are called from `board_lp.c`, it is possible to give to `PLATFORM_SetRamBanksRetained()` a different `bank_mask` adapted to your specific application.

In deep power down, this feature does not have any impact because in this power mode, all RAM banks are already powered off.

3 - Low power modes overview PWR module API provides the capability to set low power mode constraints from various components or from the application. These constraints are provided to the SDK power manager. Upper layer (all Application code, connectivity stacks, etc.) can call directly the SDK Power Manager if it requires more advanced tuning. The PWR API can be found in `PWR_Interface.h`.

Note : ‘Upper layer’ signifies all layers, applications, components, or modules that are above the connectivity framework in the Software architecture.

Note : Each power domain has its own Low Power mode capability. The Low Power modes described below are for the main domain and it is supposed that the wake up domain goes to the same Low Power mode. This is not always true as the wake up domain that contains some wake up peripheral can go a lower Low Power mode state than the main domain so the peripherals in the wake up domain can remain operational when the main domain is in Low Power mode (deep sleep or power down modes). In this case, the context of the Hardware peripheral located in the wake up domain does not need to be saved and restored as for the peripherals located in the main domain

3.1 Wait for Interrupt (WFI) Definition

In the Wait for Interrupt (WFI) state, the CPU core is powered on, but is in an idle mode with the clock turned OFF.

Wake up time and typical use case

The wakeup time from this Low Power mode is insignificant because the Fast clock from FRO is still running.

This Low Power mode is mainly used when there is an hardware activity while the Software runs the Idle task. This allows the code execution to be temporarily suspended, thus reducing a bit the power consumption of the device by switching off the processor clock. When an interrupt fires, the processor clock is instantaneously restored to process the Interrupt Service Routine (ISR).

Usage

In order to prevent the software from programming the device to go to a lower Low Power mode (such as Deep Sleep, Power Down mode or Deep Power Down mode), the component responsible for the hardware drivers shall call `PWR_SetLowPowerModeConstraint(PWR_WFI)` function. When the Hardware activity is completed, the component shall release the constraint by calling `PWR_ReleaseLowPowerModeConstraint(PWR_WFI)`.

Alternatively, the component can call `PWR_LowPowerEnterCritical()` and then `PWR_LowPowerExitCritical()` functions.

For fine tuning of the Low Power mode allowing more power saving, the component can call directly the SDK power manager API with `PM_SetConstraints()` function using the appropriate Low Power mode and low power constraint. However, this is reserved for more advanced user that knows the device very well. It is not recommended to do so.

The PWR module has no external dependencies, so the low-power entry and exit callback functions must be defined by the user for each peripheral that has specific low power constraints. It is consequently convenient to register to the component the low power callbacks structure that is used for entering and exit low power critical sections. In Bluetooth LE, you can take the example in the `app_conn.c` file as shown here :

```
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
static const SecLib_LowpowerCriticalCBs_t app_LowpowerCriticalCBs =
{
    .SecLibEnterLowpowerCriticalFunc = &PWR_LowPowerEnterCritical,
    .SecLibExitLowpowerCriticalFunc = &PWR_LowPowerExitCritical,
};
#endif

void BluetoothLEHost_Init(..)
{
    ...
    /* Cryptographic hardware initialization */
    SecLib_Init();
    #if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
    /* Register PWR functions into SecLib module in order to disable device lowpower
       during SecLib processing. Typically, allow only WFI instruction when
       commands (key generation, encryption) are processed by SecLib */
    SecLib_SetLowpowerCriticalCb(&app_LowpowerCriticalCBs);
    #endif
    ...
}
```

Limitations

No limitation when using the WFI mode.

3.2 Sleep mode Sleep mode is similar to WFI low power mode but with some additional clock gating. The Sleep mode is device specific, please consult the Hardware reference manual of the device for more information.

3.2 Deep Sleep mode Definition

In Deep Sleep mode, the fast clock is turned off, and the CPU along with the main power domain are placed into a retention state, with the voltage being scaled down to support state retention only. Because no high frequency clock is running, the voltage applied on the power domain can be reduced to reduce leakage on the hardware logic. This reduces the overall power consumption in the Deep Sleep mode. When waking up from Deep sleep mode, the core voltage is increased back to nominal voltage and the fast clock (FRO) is turned back on, the peripheral in this domain can be reused as normal.

To save more additional power, some unused RAM banks can be powered off. This prevents from having current leakage and consequently, allow to reduce even more the power consumption in Deep Sleep mode. This is achieved by calling `PLATFORM_SetRamBanksRetained()` from low power entry function from `board_lp.c` file.

Usage

All firmware is able to implement Deep Sleep mode transparently to the application thanks to the PWR module, low power platform submodule and low power board file. This is described in the section Low-power implementation.

When entering this mode, it is recommended to turn the output pins into input mode, or high impedance to reduce leakage on the pads. This is typically done in `pin_mux.c` file, called from `board.c` file and executed from the low power callback in `board_lp.c` file. As an example, the TX line of the UART peripheral can be turned to disabled so it prevents the current from being drawn by the pad in Low Power mode.

Wake up time and typical use case

The wake up time is very fast, it takes mostly the time for the Fast FRO to start up again (couple of hundreds of microseconds) so this mode is a very good balance between power consumption in low-power mode and wake up latency and shall be used extensively in most of the use cases of the application.

Limitations

In Deep Sleep mode, the clock is disabled to the CPU and the main peripheral domain, so peripheral activity (for example, an on-going DMA transfer) is not possible in Deep Sleep mode.

3.3 Power Down mode Definition

In Power Down mode, both the clock, and power are shut off to the CPU and the main peripheral domain. SRAM is retained, but register values are lost. The SDK power manager handles the restore of the processor registers and dependencies such as interrupt controller and similar ones transparently from the application.

Usage

The application, with the help of the low power board files, saves and restores the peripherals that were located in the power domain during the entry and exit of the power down mode. This is done from low power board `_lp` files in the entry/exit low power callbacks. Example is given for the serial manager and debug console in `board_lp.c` file in function `BOARD_ExitPowerDownCb()`.

If the device contains a dedicated wake up power domain where some wake up peripherals are located, if this wake up domain is not turned into power down mode but only Deep sleep mode or active mode, this peripheral does not need for a save and restore on low power entry/exit. For instance, on KW45, This is basically achieved when enabling the wakeup source of the peripheral `PWR_EnableWakeUpSource()` from `APP_ServiceInitLowpower()` function. Alternatively, this can be directly achieved by setting the constraint to the SDK power manager by calling `PM_SetConstraints()`, (use `APP_LPUART0_WAKEUP_CONSTRAINTS` for wakeup from UART constraint).

On exit from low power, The low power state of power domain can be retrieved by Platform API `PLATFORM_GetLowpowerMode()`. This API shall be called from low power exit callback function only.

As for Deep Sleep mode, software shall configure the output pins into input or high impedance during the Low Power mode to avoid leakage on the pads.

Wake up time and typical use case

The wake up time is significantly longer than wake up time from Deep Sleep (from several hundreds of micro-seconds to a couple of milliseconds depending on the platform). On some platform, it can take longer; for instance, if ROM code is implemented and perform authentication checks for security and hardware logic in power domain needs to be restored (case for KW45).

However, After ROM code execution, the SDK power manager resumes the Idle task execution from where it left before entering low-power mode. Hence, the wakeup time from this mode is still significantly lower than the initialization time from a power on reset or any other reset.

Depending on the wakeup time of the platform and the low power time duration, This mode is recommended when no Software activity is expected to happen for the next several seconds. In Bluetooth LE, this mode is preferred in advertising or without Bluetooth LE activity. However, in scanning or connected mode, Regular wakes up happens regularly for instance to retrieve HCI message responses from the Link layer, the Deep Sleep mode is rather recommended.

Limitations

In addition to the Deep Sleep limitation (no Hardware processing on going when going to Power down mode) and the significant increase of the wake time, the Power Down mode requires the ROM code to execute and this last uses significant amount of memory in SRAM.

Typically, The first SRAM bank (16 KBytes) is used by the ROM code during execution so the Application firmware can use this section of SRAM for storing bss, rw data, or stacks. Only temporary data could be stored here and this location is overwritten on every Power Down exit sequence.

In order to avoid placing firmware data section (bss, rw, etc.) in the first SRAM bank, the linker script variable `gLowpowerPowerDownEnable_d` should be set to 1. Setting the linker script variable to avoid placing firmware data section in the first SRAM bank, The effect of setting this flag is to prevent the firmware from using the first 16 KB in SRAM.

Note : This setting is ONLY required if the application implements Power Down mode. If Application uses other low-power mode, this is not required.

3.4 Deep Power-down mode Definition

In Deep Power Down mode, the SRAM is not retained. This power mode is the lowest disponible, it is exited through reset sequence.

Usage

In addition to the Power Down limitation, the Deep Power Down mode shut down all memory in SRAM. Because it is exited through reset sequence the wake time is also longer.

Wake up time and typical use case

As this low-power mode is exited through the reset sequence, the wake up time is longer than any other mode. In Bluetooth LE, this mode is possible in no Bluetooth LE activity, and is preferred if we know that there will be no Bluetooth LE activity before a several amount of time.

Limitations

All memory in SRAM will be shut down in deep power down, the main limitation in going in this low-power mode is that the context will not be saved.

ModuleInfo

Overview The ModuleInfo is a small Connectivity Framework module that provides a mechanism that allows stack components to register information about themselves.

The information comprises :

- Component or module name (for example: Bootloader, IEEE 802.15.4 MAC, and Bluetooth LE Host) and associated version string
- Component or module ID
- Version number
- Build number

The information can be retrieved using shell commands or FSCI commands.

Detailed data types and APIs used in ConnFWK_APIs_documentation.pdf.

NVM: Non-volatile memory module

Overview In a standard Harvard-architecture-based MCU, the flash memory is used to store the program code and program constant data. Modern processors have a built-in flash memory controller that can be used under user program execution to store non-volatile data. The flash memories have individually erasable segments (sectors) and each segment has a limited number of erase cycles. If the same segments are used to store various kinds of data all the time, those segments quickly become unreliable. Therefore, a wear-leveling mechanism is necessary to prolong the service life of the memory. The NVM module in the connectivity framework provides a file system with a wear-leveling mechanism, described in the subsequent sections. The *NvIdle()* function handles the program and erase memory operations. Before resetting the MCU, *NvShutdown()* must be called to ensure that all save operations have been processed.

NVM boundaries and linker script requirement Most of the MCUs have only a standard flash memory that the non-volatile (NV) storage system uses. The amount of memory that the NV system uses for permanent storage and its boundaries are defined in the linker configuration file though the following linker symbols :

- NV_STORAGE_START_ADDRESS
- NV_STORAGE_END_ADDRESS
- NV_STORAGE_MAX_SECTORS
- NV_STORAGE_SECTOR_SIZE

The reserved memory consists of two virtual pages. The virtual pages are equally sized and each page is using one or more physical flash sectors. Therefore, the smallest configuration is using two physical sectors, one sector per virtual page.

NVM Table The Flash Management and Non-Volatile Storage Module holds a pointer to a RAM table. The upper layers of this table register information about data that the storage system should save and restore. An example of NVM table entry list is given below.

pData	ElemCount	ElemSize	EntryId	EntryType
0x1FFF9000	3	8	0xF1F4	MirroredInRam
0x1FFF7640	5	4	0xA2A6	NotMirroredInRam
0x1FFF1502	6	1	0x4212	NotMirroredInRam AutoRestore
0x1FFFF200	2	6	0x118F	MirroredInRam

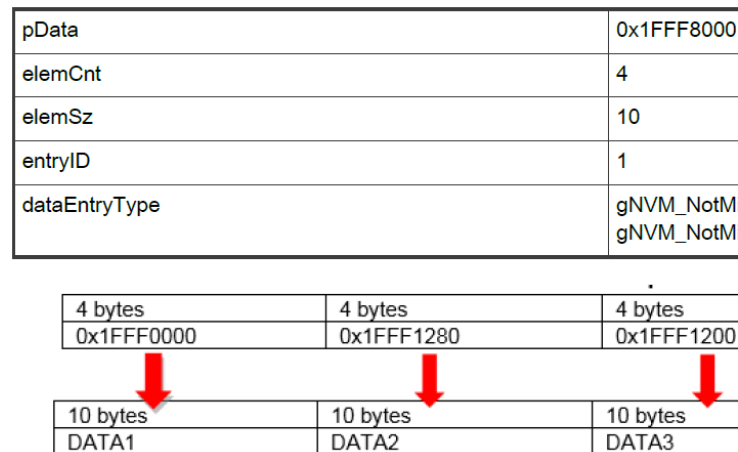
NVM Table entry As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

A RAM table entry has the following structure:

- pData (4 bytes) is a pointer to the RAM memory location where the dataset elements are stored.

- elemCnt (2 bytes) represents how many elements the dataset has.
- elemSz (2 bytes) is the size of a single element.
- entryID is a 16-bit unique ID of the dataset.
- dataEntryType is a 16-bit value representing the type of entry (mirrored/unmirrored/unmirrored auto restore).

For mirrored datasets, pData must point directly to the RAM data. For unmirrored datasets, it must be a double pointer to a vector of pointers. Each pointer in this table points to a RAM/FLASH area. Mirrored datasets require the data to be permanently kept in RAM, while unmirrored datasets have dataset entries either in flash or in RAM. If the unmirrored entries must be restored at the initialization, NotMirroredInRamAutoRestore should be used. The entryID gUnmirroredFeatureSet_d should be set to 1 for enabling unmirrored entries in the application. The last entry in the RAM table must have the entryID set to gNvEndOfTableId_c.



The figure below provides an example of table entry :

When the data pointed to by the table entry pointer (pData) has changed (entirely or just a single element), the upper layers call the appropriate API function that requests the storage system to save the modified data. All the save operations (except for the synchronous save and atomic save) and the page erase and page copy operations are performed on system idle task. The application must create a task that calls NvIdle in an infinite loop. It should be created with OSA_PRIORITY_IDLE. However, the application may choose another priority. The save operations are done in one virtual page, which is the active page. After a save operation is performed on an unmirrored dataset, pData points to a flash location and the RAM pointer is freed. As a result, the effective data should always be allocated using the memory management module.

Active page The active page contains information about the records and the records. The storage system can save individual elements of a table entry or the entire table entry. Unmirrored datasets can only have individual saves. On mirrored datasets, the save/restore functions must receive the pointer to RAM data. For example, if the application must save the third element in the above vector, it should send $0x1FFF8000 + 2 * \text{elemSz}$. For unmirrored datasets, the application must send the pointer that points to the area where the data is located. For example, if the application must save the third element in the above vector, it should send $0x1FFF8000 + 2 * \text{sizeof(void*)}$.

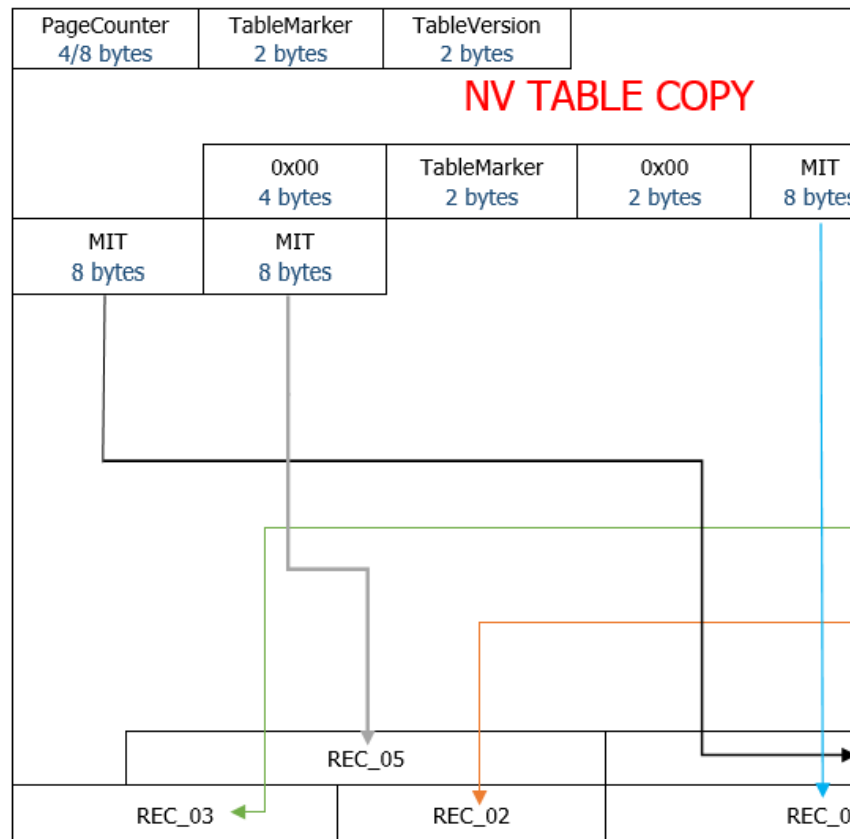
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page erase operation is performed when the system is formatted. It is also performed when the page is full and a new record cannot be written into that page. Before being erased, the full page is first copied (only the most recent saves) and erased afterward.

The validity of the Meta Information Tag (MIT), and, therefore, of a record, is guaranteed by the MIT start and stop validation bytes. These two bytes must be equal to consider the record

referred by the MIT valid. Furthermore, the value of these bytes indicates the type of the record, whether it is a single element or an entire table entry. The nonvolatile storage system allows dynamic changes of the table within the RAM memory, as follows:

- Remove table entry
- Register table entry

A new table entry can be successfully registered if there is at least one entry previously removed or if the NV table contains uninitialized table entries, declared explicitly to register new table entries at run time. A new table entry can also replace an existing one if the register table entry is called with an overwrite set to true. This functionality is disabled by default and must be enabled by the application by setting `gNvUseExtendedFeatureSet_d` to 1.



The layout of an active page is shown below:

As shown above, the table stored in the RAM memory is copied into the flash active page, just after the table version. The “table start” and “table end” are marked by the table markers. The data pointers from RAM are not copied. A flash copy of a RAM table entry has the following

entryId	entryType	elemCnt	elemSz
2 bytes	2 bytes	2 bytes	2 bytes

structure:

Where:

- entryID is the ID of the table entry
- entryType represents the type of the entry (mirrored/unmirrored/unmirrored auto restore)
- elemCnt is the elements count of that entry
- elemSz is the size of a single element

This copy of the RAM table in flash is used to determine whether the RAM table has changed. The table marker has a value of 0x4254 (“TB” if read as ASCII codes) and marks the beginning

and end of the NV table copy.

After the end of the RAM table copy, the Meta Information Tags (MITs) follow. Each MIT is used to store information related to one record. An MIT has the following structure:

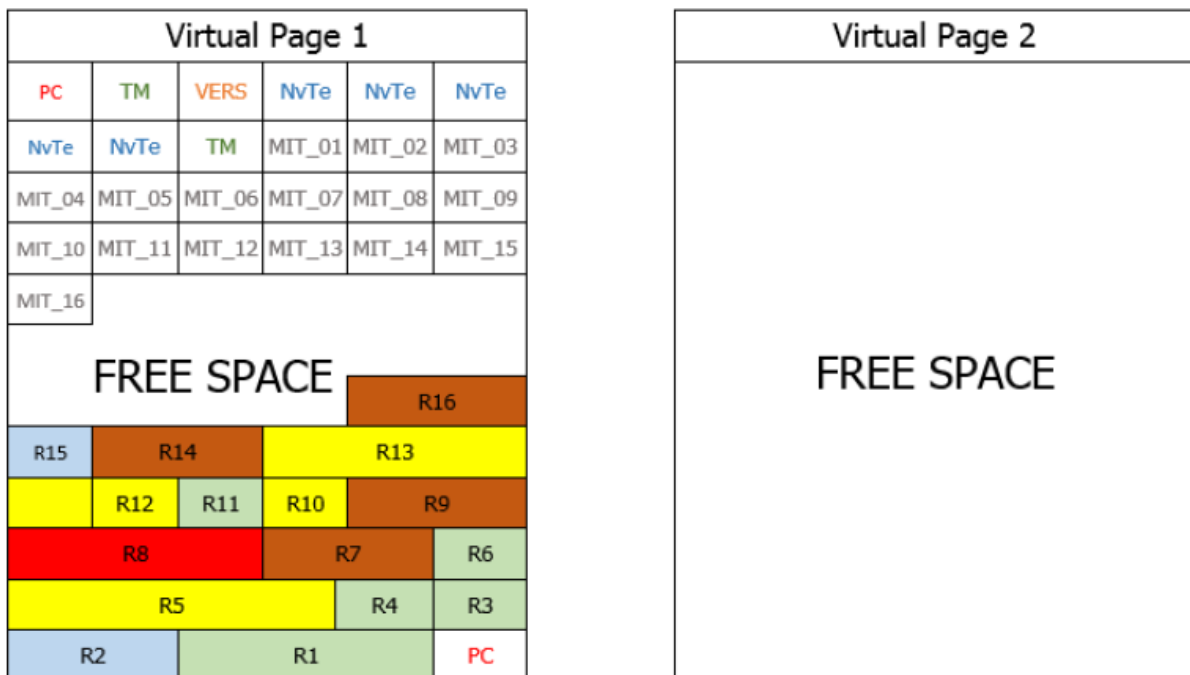
VSB	entryID	elemIdx	recordOffset	VEB
1 byte	2 bytes	2 bytes	2 bytes	

Where:

- VSB is the validation start byte.
- entryID is the ID of the NV table entry.
- elemIdx is the element index.
- recordOffset is the offset of the record related to the start address of the virtual page.
- VEB is the validation end byte.

A valid MIT has a VSB equal to a VEB. If the MIT refers to a single-element record type, VSB=VEB=0xAA. If the MIT refers to a full table entry record type (all elements from a table entry), VSB=VEB=0x55. Because the records are written to the flash page, the available page space decreases. As a result, the page becomes full and a new record does not have enough free space to be copied into that page.

In the example given below, the virtual page 1 is considered to be full if a new save request is pending and the page free space is not sufficient to copy the new record and the additional MIT. In this case, the latest saved datasets (table entries) are copied to virtual page 2.

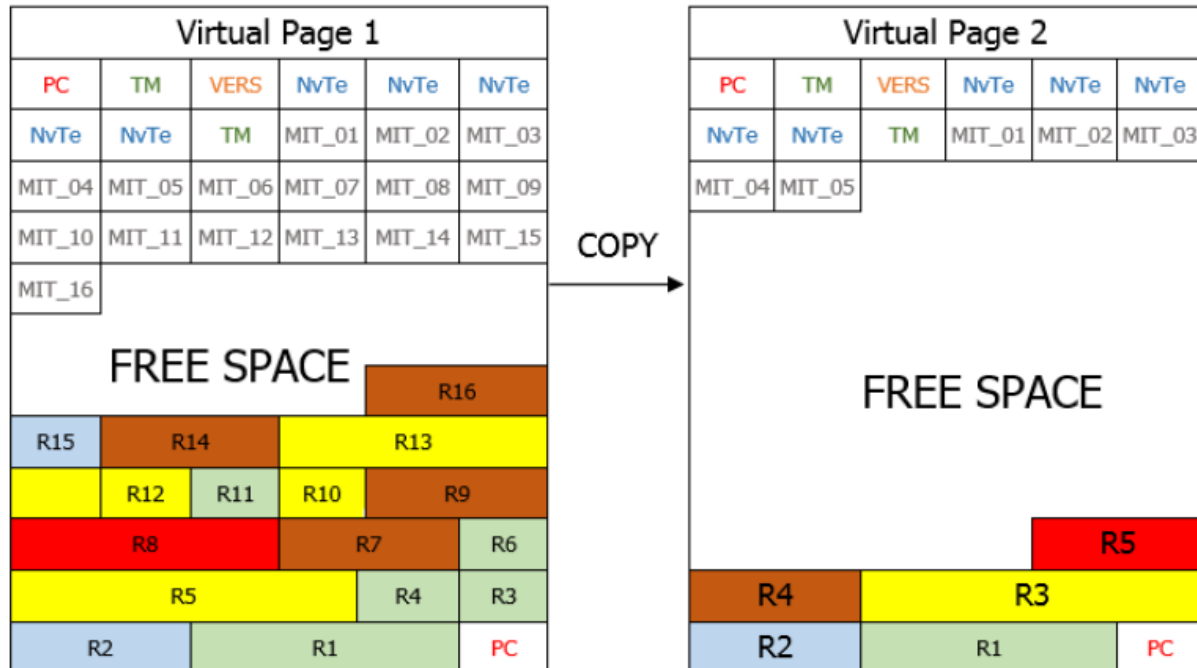


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- R1 is a 'full' record type (contains all the NV table entry elements), whereas R3, R4, R6 and R11 are 'single' record types.
- R2 – full record type; R15 – single record type
- R5, R13 – full record type; R10, R12 – single record type

- R8 – full record type
- R7, R9, R14, R16 – full record type

As shown above, the R3, R4, R6, and R11 are ‘single’ record types, while R1 is a ‘full’ record type of the same dataset. When copied to virtual page 2, a defragmentation process takes place. As a result, the record copied to virtual page 2 has as much elements as R1, but individual elements are taken from R3, R4, R6, and R11. After the copy process completes, the virtual page 2 has five ‘full’ record types, one for each dataset. | This is illustrated below:



Finally, the virtual page 2 is validated by writing the PC value and a request to erase virtual page 1 is performed. The page is erased on an idle task, sector by sector where only one sector is erased at a time when idle task is executed.

If there is any difference between the RAM and flash tables, the application must call `RecoverNvEntry` for each entry that is different from its RAM copy to recover the entry data (ID, Type, ElemSz, ElemCnt) from flash before calling `NvInit`. The application must allocate the `pData` and change the RAM entry. It can choose to ignore the flash entry if the entry is not desired. If any entry from RAM differs from its flash equivalent at initialization, a page copy is triggered that ignores the entries that are different. In other words, data stored in those entries is lost.

The application can check if the RAM table was updated. In other words, if the MCU program was changed and the RAM table was updated, using the function `GetFlashTableVersion` and compare the result with the constant `gNvFlashTableVersion_c`. If the versions are different, `NvInit` detects the update and automatically upgrades the flash table. The upgrade process triggers a page copy that moves the flash data from the active page to the other one. It keeps the entries that were not modified intact and it moves the entries that had their elements count changed as follows:

- If the RAM element count is smaller than the flash element count, the upgrade only copies as many elements as are in RAM.
- If the RAM element count is larger than the flash element count, the upgrade copies all data from flash and fills the remaining space with data from RAM. If the entry size is changed, the entry is not copied. Any entryIds that are present in flash and not present in RAM are also not copied. This functionality is not supported if `gNvUseExtendedFeatureSet_d` is not set to 1.

ECC Fault detection The KW45/K32W1 internal flash is organized in 16 byte phrases and 8kB sectors (minimal erase unit). Its flash controller is synthesized so that it generates ECC information and an ECC generator / checker. During the programming of internal flash, errors may accidentally happen and cause ECC errors as a flash phrase is being written. These may happen due to multiple reasons:

- programmatic errors such as overwriting an already programmed phrase (transitioning bits from 0b to 1b). These are evitable by performing a blank check verification over phrase to be programmed, at the expense of processing power.
- occurrence of power drop or glitches during a programming operation.
- excessive wear of flash sector. The flash controller is capable of correcting one single ECC error but raises a bus fault whenever reading a phrase containing more than one ECC fault. Once an ECC error has ‘infected’ a flash phrase, the fault will remain and raise again at each read operation over the same phrase including blank check and prefetch. It can only be rid of by erasing the whole flash sector that contained the faulty phrase. In order to recover from situations where an ECC fault has occurred a `gNvSalvageFromEccFault_d` option has been added, which forces `gNvVerifyReadBackAfterProgram_d` to be defined to TRUE. If defined, the `gNvVerifyReadBackAfterProgram_d` option of the NVM module, causes the program to read back the programmed area after every flash programming operation. The verification is performed in safe mode if `gNvSalvageFromEccFault_d` is also defined. This is so as to detect ECC faults as early as possible as they appear, indeed when verifying a programming operation, one cannot be certain of the absence of ECC fault and avoid the bus fault. The safe API is thence used to perform the read back operation is performed using this safe API, so that we can tread in the flash and detect potential errors. The defects are detected on the fly whereas in the absence of safe read back, the error would cause a fault, potentially much later. During normal operation, assuming that no chip reset was provoked, this will consist in a single ECC fault either in the last record data or its meta information. Detecting such a fault calls for an immediate page copy to the other virtual page, so that the currently active page gets erased and the error gets cleared. Should the ECC fault occurs in the middle of a page copy operation, the switch of active page is postponed so that the fault page can be erased again and the copy can be restarted.

If the system underwent a power drop during a flash programming operation, sufficient to provoke a reset, at the ensuing reboot, ECC fault(s) may be present in the NVM area at the location that was being written. The detection is performed by an NVM sweeping mechanism, using the safe read API. That marks the faulty virtual page so that all subsequent reads within this virtual page are done with the safe API. If this case arises, a copy of the valid contents of the faulty page is attempted to the other virtual page. At NVM initialization, faults should be detected, either at the top of the meta data or at the bottom of the record area within the previous active page. This should guarantee that only the latest record write operation may be impaired. When the page copy has taken place, the faulty page is erased and the execution may resume. During `NvCopyPage`, when ‘garbage collecting’ occurs or whenever the current virtual active page needs to be transferred to the other virtual page, ECC errors are intercepted so that the operation can be attempted again in case of error. In case of NVM contents clobbering by programming errors, the salvage operation does its best to rescue as many records as possible but data will inevitably be lost.

An additional option -namely `gInterceptEccBusFaults_d` - was introduced in order to catch and correct ECC faults at Bus Fault handler level. Indeed, should an ECC bus fault fire, in spite of the precautions taken with NVM’s `gNvSalvageFromEccFault_d`, we verify if the fault belongs to the NV storage. If so, a drastic policy can be adopted consisting in an erasure of the faulty sector. The corresponding Bus Fault handling is not part of the NVM, but dwells in the framework platform specific sources. Alternative handling could be implemented by the customer.

Save policy: Execution of program and erase operations on a flash an MCU core fetches code from cause perturbations of the core activity or requires to place critical code in RAM so that real-time ISR can still be served. The penalty of a sector erase is much higher than a simple program operation. The NVM is designed so as to limit the erase operations at ‘garbage collecting’ time,

so that flash wear is limited and no time is wasted. Several write policies are implemented to cope with the application constraints, one synchronous mode API and several posted write APIs. Among the posted write policies, the `gNvmSaveOnIdleTimerPolicy_d` compilation option selects a mode where flash write operations occur at time interval within the Idle task. Another option exists to ‘randomize’ the time interval with some jitter.

- 1) `NvSyncSave` performs a write synchronously with the disadvantage of stalling processor activity until comp
- 2) `NvSaveOnCount` posts a pending write operation and postpones the actual flash operation until number of record updates has reached a maximum. The actual write happens during Idle Task execution. see `NvSetCountsBetweenSaves` related API.
- 3) `NvSaveOnInterval`: posts a pending write operation and postpones the actual flash operation until the predefined number of ticks has elapsed. Optional mode - Active if (`gNvmSaveOnIdleTimerPolicy_d` & `gNvmUseSaveOnTimerOn_c`). see `NvSetMinimumTicksBetweenSaves` related API. Note that `gNvmUseSaveIntervalJitter_c` policy is a sub-option of `gNvmSaveOnIdleTimerPolicy_d` used to randomize slightly the time at which the write operation will happen.

Constant macro definition

- `gNvStorageIncluded_d` : If set to TRUE, it enables the whole functionality of the nonvolatile storage system. By default, it is set to FALSE (no code or data is generated for this module).
- `gNvUseFlexNVM_d` : If set to TRUE, it enables the FlexNVM functionality of the nonvolatile storage system. By default, it is set to FALSE. If FlexNVM is used, the standard nonvolatile storage system is disabled.
- `gNvFragmentation_Enabled_d` : Macro used to enable/disable the fragmented saves/restores (a particular element from a table entry can be saved or restored). It is set to FALSE by default.
- `gNvUseExtendedFeatureSet_d` : Macro used to enable/disable the extended feature set of the module:
 - Remove existing NV table entries
 - Register new NV table entries
 - Table upgradeIt is set to FALSE by default.
- `gUnmirroredFeatureSet_d` : Macro used to enable unmirrored datasets. It is set to 0 by default.
- `gNvTableEntriesCountMax_c` : This constant defines the maximum count of the table entries (datasets) that the application is going to use. It is set to 32 by default.
- `gNvRecordsCopiedBufferSize_c` : This constant defines the size of the buffer used by the page copy function, when the copy operation performs defragmentation. The chosen value must be bigger than the maximum number of elements stored in any of the table entries. It is set by default to 64.
- `gNvCacheBufferSize_c` : This constant defines the size of the cache buffer used by the page copy function, when the copy operation does not perform defragmentation. The chosen value must be a multiple of 8. It is set by default to 64.
- `gNvMinimumTicksBetweenSaves_c` : This constant defines the minimum timer ticks between dataset saves (in seconds). It is set to 4 by default.
- `gNvCountsBetweenSaves_c` : This constant defines the number of calls to ‘`NvSaveOnCount`’ between dataset saves. It is set to 256 by default.

- *gNvInvalidDataEntry_c* : Macro used to mark a table entry as invalid in the NV table. The default value is 0xFFFFFU.
- *gNvFormatRetryCount_c* : Macro used to define the maximum retries count value for the format operation. It is set to 3 by default.
- *gNvPendingSavesQueueSize_c* : Macro used to define the size of the pending saves queue. It is set to 32 by default.
- *gFifoOverwriteEnabled_c* : Macro used to enable overwriting older entries in the pending saves queue (if it is full). If it is FALSE and the queue is full, the module tries to process the oldest save in the queue to free a position. It is set to FALSE by default.
- *gNvMinimumFreeBytesCountStart_c* : Macro used to define the minimum free space at initialization. If the free space is smaller than this value, a page copy is triggered. It is set by default to 128.
- *gNvEndOfTableId_c* : Macro used to define the ID of the end-of-table entry. It is set to 0xFFFEU by default. No valid entry should use this ID.
- *gNvTableMarker_c* : Macro used to define the table marker value. The table marker is used to indicate the start and the end of the flash copy of the NV table. It is set to 0x4254U by default.
- *gNvFlashTableVersion_c* : Macro used to define the flash table version. It is used to determine if the NVM table was updated. It is set to 1 by default. The application should modify this every time the NVM table is updated and the data from NVM is still required.
- *gNvTableKeptInRam_d* : Set *gNvTableKeptInRam_d* to FALSE, if the NVM table is stored in FLASH memory (default). If the NVM table is stored in RAM memory, set the macro to TRUE.
- *gNvVerifyReadBackAfterProgram_d* : set by default force verification of NVM programming operations. Is forced implicitly when *gNvSalvageFromEccFault_d* is defined.
- *gNvSalvageFromEccFault_d* : use safe flash API to read from flash, and provide corrective action when ECC fault is met.

OtaSupport: Over-the-Air Programming Support

Overview This module includes APIs for the over-the-air image upgrade process. A Server device receives an image over the serial interface from a PC or other device thorough FSCI commands. If the Server has an image storage, the image is saved locally. If not, the image is requested chunk by chunk: With image storage

- *OTA_RegisterToFsci()*
- *OTA_InitExternalMemory()*
- *OTA_WriteExternalMemory()*
- ...
- *OTA_WriteExternalMemory()*

Without image storage:

- *OTA_RegisterToFsci()*
- *OTA_QueryImageReq()*
- *OTA_ImageChunkReq()*
- ...
- *OTA_ImageChunkReq()*

A Client device processes the received image by computing the CRC and filter unused data and stores the received image into a non-volatile storage. After the entire image has been transferred and verified, the Client device informs the Bootloader application that a new image is available, and that the MCU must be reset to start the upgrade process. See the following command sequence:

- `OTA_StartImage()`
- `OTA_PushImageChunk()` and `OTA_CrcCompute ()`
- ...
- `OTA_PushImageChunk()` and `OTA_CrcCompute ()`
- `OTA_CommitImage()`
- `OTA_SetNewImageFlag()`
- `ResetMCU()`

SecLib_RNG: Security library and random number generator

SecLib Implementation Flavors

Overview The NXP Connectivity Framework provides multiple SecLib implementations (flavors) to support different hardware capabilities, security requirements, and performance needs. Each flavor offers the same high-level cryptographic APIs but with different underlying implementations optimized for specific use cases.

Available SecLib Flavors

1. Software SecLib (SecLib.c)(RNG.c) **Description:** Pure software implementation providing cryptographic operations if no hardware is available.

Platforms: all platforms

Characteristics:

- Platform-independent software implementation
- No hardware acceleration requirements
- Consistent behavior across all platforms
- Higher CPU usage and power consumption
- Suitable for platforms without crypto hardware

Files: `SecLib.c`, `SecLib_aes_mmo.c`, `RNG.c`

2. EdgeLock SecLib (SecLib_ess.c)(RNG.c) **Description:** Hardware-accelerated implementation using NXP's EdgeLock secure subsystem s200.

Platforms: devices with S200: KW45, KW47, MCXW71, MCXW72

Characteristics:

- Hardware-backed security operations
- Secure key storage in hardware
- Hardware random number generation

- Lower CPU usage, better performance
- Enhanced security through hardware isolation
- Platform-specific (requires Secure Subsystem API)

Files: SecLib__sss.c, RNG.c

3. PSA Crypto SecLib (SecLib_psa.c)(RNG_psa.c) **Description:** Implementation based on ARM's Platform Security Architecture (PSA) Crypto API standard.

Platforms: MCXW23, (MCXW71 and MCXW72 for evaluation)

Characteristics:

- Industry-standard API
- Hardware acceleration when available
- Secure key management
- Future-proof and portable
- Compliance-ready

Files: SecLib_psa.c, SecLib_psa_config.h RNG_psa.c

4. MbedTLS SecLib (SecLib_mbedtls.c)(RNG_mbedtls.c) **Description:** Implementation using the MbedTLS cryptographic library.

Platforms: all platforms, RW61

Characteristics:

- Well-tested and widely adopted
- Software and hardware acceleration support
- Extensive algorithm support
- Good performance optimization
- Large memory footprint

Files: SecLib_mbedtls.c, SecLib_mbedtls_config.h, RNG_mbedtls.c

Flavor Selection SecLib flavor is described in '*Kconfig*' and selected inside prj.conf or defconfig. Define one of this line to override the default configuration for your board:

```
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.sw=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.secure_subsystem=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.psa=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.mbedtls=y
```

The necessary configuration files are automatically added inside of CmakeLists.txt

Random number generator

Overview The RNG module is part of the framework used for random number generation. It uses hardware RNG peripherals as entropy sources (TRNG, Secure Subsystem, ...) to provide a true random number generator interface. A Pseudo-Random number generator (PRNG) implementation is available. The PRNG may depend of SecLib services (thus requiring a common mutex) to perform HMAC-SHA256, SHA256, AES-CTR, or alternatively a Lehmer Linear Congruential generator. A prerequisite for the PRNG to function with desired randomness is to be seeded using a proper source of entropy. If no hardware acceleration is present, the RNG may fallback to lesser quality ad-hoc source e.g if present SIM_UID registers, the UIDL is used as the initial seed for the random number generator.

Initialization The RNG module requires an initialization via a call to `RNG_Init`. The RNG initialization involves a call to `RNG_SetSeed`.

In the case of a dual core system consisting of a Host core and an NBU core, the Secure Subsystem is owned by the Host core. The Host core then has a direct access to its TRNG embedded in its secure subsystem. On the NBU code side, a request is emitted via RPMSG to the Host to provide a seed. On receipt of this request, the Host sets a 'reseed needed' flag (from the ISR context) If the core running the RNG service owns the TRNG entropy hardware (if any), it can get the seed directly from this hardware synchronously. In the case of an NBU that does not control the devices entropy source, that is owned by the Host, it request a seed from the Host processor via RPMSG exchange. On receipt of this request the Host sets a flag notifying of this request from the RPMSG ISR context.

`RNG_ReInit` API is to be used at wake up time in the context of LowPower. `RNG_DeInit` is used for unit tests and coverage purposes but has no useful role in a real application.

Asynchronous Seed Handling Enhancement The RNG module has been enhanced to use the framework's work queue system instead of polling from the IDLE thread.

Key Changes:

- **Work Queue Integration:** `RNG_NotifyReseedNeeded()` now submits seed operations to the work queue on Host cores
- **Non-blocking:** Replaces previous IDLE thread polling with asynchronous work queue scheduling
- **ISR-Safe:** Seed requests from ISR context are safely deferred to work queue execution

Benefits:

- Improved system responsiveness by removing IDLE thread dependency
- Better resource management through work queue scheduling

Seed handling `RNG_SetSeed`: `RNG_SetExternalSeed` may be used to inject application entropy to RNG context seed using a supplied array of bytes. `RNG_IsReseedNeeded` used from task in Host core to check whether seed must be sent to NBU core.

`RNG_GetTrueRandomNumber` is the API used to generate a Random 32 bit number from a HW source of entropy. It is essential if only to seed the pseudo random number generator.

`RNG_GetPseudoRandomData` is used to generate arrays of random bytes.

`RNG_NotifyReseedNeeded` Enhanced to use work queue submission on Host cores instead of requiring IDLE thread polling. Sets reseed flag and schedules asynchronous seed generation.

Security Library

Overview The framework provides support for cryptography in the security module. It supports both software and hardware encryption. Depending on the device, the hardware encryption uses either the S200, MMCAU, LTC, or CAU3 module instruction set or dedicated AES and SHA hardware blocks.

Software implementation is provided in a library format.

Support for security algorithms

	SW SecLib : SecLib.c	EdgeLock SecLib_sss.c	SecLib_p	Mbedtls SecLib_mbec	nccl (part of SecLib.c)	Usage example
AES_128	SecLib_aes.c	x	x	x		
AES_128_ECB		x	x	x		
AES_128_CBC	x	x		x		
AES_128_CTR encryption	x	x				
AES_128_OFB encryption	x					
AES_128_CMAC	x	x	x	x		BLE connection, ieee 15.4
AES_128_EAX	x					
AES_128_CCM	x	x	x	x		BLE, ieee 15.4
SHA1	SecLib_sha.c	x		x		
SHA256	x	x	x	x		
HMAC_SHA256	x	x		x		PRNG, Digest for Matter
ECDH_P256 shared secret generation	x (by 15 incremental steps) -> SecLib_ecdh.c	x with MACRO SecLibECDHUseSSS	x	x	x	BLE pairing,
EC_P256 key pair generation	x	x	x	x	x	
EC_P256 public key generation from private key				x	x	Matter (ECDSA)
ECDSA_P256 hash and msg signature generation / verification		only if owner of the key pair		x	x	Matter
SPAKE2+ P256 arithmetics				x	x	Matter

SecLib_psa The framework provides PSA (Platform Security Architecture) Crypto API support through the SecLib_psa.c implementation. This provides a standardized cryptographic interface

that can leverage hardware acceleration when available.

Configuration PSA support is configured through the `SecLib_psa_config.h` file, which defines:

- Hardware acceleration capabilities via `MBEDTLS_PSA_ACCEL_*` macros
- Memory management integration with the framework's memory manager
- External RNG integration

Advantages of PSA Crypto API

Standardization and Portability

- **Vendor Independence:** Applications using PSA APIs can be easily ported between different hardware platforms without changing cryptographic code
- **Industry Standard:** PSA Crypto API is an ARM-defined industry standard (PSA Certified), ensuring consistent behavior across different platforms and vendors
- **Future-Proof:** As an evolving standard, PSA provides long-term compatibility and support for emerging cryptographic requirements

Security Benefits

- **Secure Key Management:** PSA provides built-in secure key storage and lifecycle management, reducing the risk of key exposure
- **Hardware Security Integration:** Seamless integration with hardware security modules (HSMs) and secure elements when available
- **Isolation:** Keys and cryptographic operations can be isolated from application code, reducing attack surface
- **Tamper Resistance:** Hardware-backed implementations provide protection against physical attacks

BLE advanced secure mode

New elements in existing structures: `computeDhKeyParam_t::keepInternalBlob` - boolean telling if the shared blob is kept in this structure (in `.outpoint`) after `ECDH_P256_ComputeDhKey()` or `ECDH_P256_ComputeDhKeySeg()` call.

New arguments in existing functions: `ECDH_P256_ComputeDhKey` `keepBlobDhKey` - boolean telling `ECDH_P256_ComputeDhKey()` or `ECDH_P256_ComputeDhKeySeg()` to keep the shared object after computation for later use (it is required by the `SecLib_GenerateBluetoothF5KeysSecure`).

New macros: `gSecLibSssUseEncryptedKeys_d` - Enable or disable S200 blobs SecLib support. 0 - the Bluetooth Keys are available in plaintext, 1 - the Bluetooth Keys are not available in plaintext, but in secured blobs. Default is disabled.

New functions:

LE Secure connections pairing:

void ECDH_P256_FreeDhKeyDataSecure This is a function used to free the shared object stored in computeDhKeyParam_t. When user calls ECDH_P256_ComputeDhKeySeg() with keepBlobDhKey set to 1, it should also call **ECDH_P256_FreeDhKeyDataSecure**.

SecLib_GenerateBluetoothF5Keys This function is extracted from the Bluetooth LE Host Stack implementation. This corresponds to the legacy implementation without key blobs.

SecLib_GenerateBluetoothF5KeysSecure Similar to **SecLib_GenerateBluetoothF5Keys** this function is modified to work with key blobs, the reason is to not use SSS inside the Bluetooth LE Host Stack.

SecLib_DeriveBluetoothSKD This is a helper function used by the Bluetooth LE Host Stack in the pairing procedure, when receiving the vendor HCI command specifying that the ESK needs to be provided to LL.

ELKE_BLE_SM_F5_DeriveKeys This is a private function, helper for **SecLib_GenerateBluetoothF5KeysSecure**. It was provided by the STEC team.

Privacy:

SecLib_ObfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to obfuscate the IRK before setting it to Bluetooth LE Controller or before saving it to NVM

SecLib_DeobfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to extract the plaintext IRK key from the saved NVM blob.

SecLib_VerifyBluetoothAh This function is extracted from the legacy Bluetooth LE Host Stack implementation using plaintext keys.

SecLib_VerifyBluetoothAhSecure Similar to **SecLib_VerifyBluetoothAh** with modification to work with S200 key blob.

SecLib_GenerateSymmetricKey This is a function used by the application to generate the local IRK and local CSRK.

SecLib_GenerateBluetoothEIRKBlobSecure This is a function used by the application to generate the EIRK needed by Bluetooth LE Controller from the IRK blob.

A2B feature

ECDH_P256_ComputeA2BKey This function is used to compute the EdgeLock to EdgeLock key. pInPeerPublicKey points to the peer public key, pOutE2EKey is the pointer to where the E2E key object will be stored, this will be freed by the application when it is no longer required by calling ECDH_P256_FreeE2EKeyData().

ECDH_P256_FreeE2EKeyData This function is used to free the key object given as a parameter. It is used by the application to free the E2E key when is no longer needed.

SecLib_ExportA2BBlobSecure This function is used to import an ELKE blob or plain text symmetric key in s200 and export an E2E key blob. The input type is identified by the keyType parameter.

SecLib_ImportA2BBlobSecure This function is used to import an E2E key blob in s200 and export an ELKE blob or plain text symmetric key. The output type is identified by the keyType parameter.

LE Secure connections Pairing flow and SecLib usage:

1. Each device needs to generate locally the public+private keypair. This is done using **ECDH_P256_GenerateKeys**.
2. Devices exchange their public keys.
3. Upon receiving the peer device's public key, local device is computing DH key using **ECDH_P256_ComputeDhKey**.
4. Each device sends DHKeyCheck packet
5. Upon receiving DhKeyCheck each device computes LTK blob using **SecLib_GenerateBluetoothF5Keys**
6. After computing the each device sends HCI_LeStartEnc (on initiator), HCI_Le_Provide_Long_Term_Key (on responder)
7. Bluetooth LE Controller sends back SKD report custom event
8. Bluetooth LE Host Stack computes ESKD based on LTK blob using **SecLib_DeriveBluetoothSKD** and sends it to Bluetooth LE Controller
9. Bluetooth LE Controller encrypts the link

IRK flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received:
 - the local IRK is generated using **SecLib_GenerateSymmetricKey**
 - the local EIRK is generated using **SecLib_GenerateBluetoothEIRKBlobSecure**
 - local CSRK is generated using **SecLib_GenerateSymmetricKey**
2. During legacy pairing when receiving bonding keys, IRK is obfuscated using **SecLib_ObfuscateKeySecure** and stored
3. When app wants to set the OOB keys using Gap_SaveKeys the IRK is obfuscated using **SecLib_ObfuscateKeySecure**
4. When application calls API Gap_VerifyPrivateResolvableAddress IRK is obfuscated using **SecLib_ObfuscateKeySecure** and verified using **SecLib_VerifyBluetoothAhSecure**
5. When a new connection is received in Host with RPA address not resolved by the Bluetooth LE Controller, the Host tries to resolve it by obfuscating it using **SecLib_ObfuscateKeySecure** and verifying it using **SecLib_VerifyBluetoothAhSecure**
6. When adding a peer in Bluetooth LE Controller resolving list, the peer's IRK is obfuscated using **SecLib_ObfuscateKeySecure** before setting it using **HCI_Le_Add_Device_To_Resolving_List**.

7. When an IRK plaintext is requested by the application using `Gap_LoadKeys` it is obtained using **`SecLib_DeobfuscateKeySecure`**
8. When legacy pairing completes and LTK needs to be send in the pairing complete event (`gConnEvtPairingComplete_c`) the **`SecLib_DeobfuscateKey`** is used to extract the plaintext.

A2B flow and SecLib usage:

1. At startup, when `gInitializationComplete_c` event is received, the application will call **`ECDH_P256_GenerateKeys`** to generate the public/private key pair required for the E2E key derivation and send the public key to the peer device.
2. When the public key is received from the peer device, the application will call **`ECDH_P256_ComputeA2BKeySecure`** to generate the EdgeLock to EdgeLock key.
3. The application will obtain an E2E IRK blob by calling **`SecLib_ExportA2BBlobSecure`** with key type `gSecElkeBlob_c`. The obtained blob is sent to the peer anchor. The peer anchor will call **`SecLib_ImportA2BBlob`** with keyType `gSecElkeBlob_c` and save the resulting ELKE blob in NVM, for Digital Key both anchors must have the same IRK.
4. After pairing, in order to send the LTK and IRK contained in the bonding data securely, the application will call **`SecLib_ExportA2BBlobSecure`** with keyType `gSecLtkElkeBlob_c` for the LTK, and **`SecLib_ExportA2BBlobSecure`** with keyType `gSecPlainText_c` for the IRK. The E2E blobs obtained are sent along with the rest of the bonding data to the peer anchor device.
5. After the bonding data is trasfered the E2E key is no longer needed and **`ECDH_P256_FreeE2EKeyData`** is called with the key object obtained at step 2 when **`ECDH_P256_ComputeA2BKeySecure`** was called.

Sensors

Overview The Sensors module provides an API to communicate with the ADC. Two values can be obtained by this module :

- Temperature value
- Battery level

The temperature is given in tenths of degrees Celsius and the battery in percentage.

This module is multi-caller, the ADC is protected by a mutex on the resource and by preventing lowpower (only WFI) during its processing. Platform specific code can be find in `fwk_platform_sensors.c/h`.

Features

Manual Measurement Calling “`SENSORS_GetTemperature()`” or “`SENSORS_GetBatteryLevel()`” function will give the last measured value that was copied to RAM. If you want to refresh this value you must first call the trigger function (`SENSORS_TriggerTemperatureMeasurement()` / `SENSORS_TriggerBatteryMeasurement()`) that configures the ADC to perform temperature or battery measurement, then a call to the refresh function (`SENSORS_RefreshTemperatureValue()` / `SENSORS_RefreshBatteryLevel()`) will read the returned value from the ADC and save it in RAM. You can run code between these two functions because the ADC won’t give you the value until it’s fully ready.

Important: When a measurement is ongoing, new measurement requests will be **dropped**. Measurement triggers are only accepted when the measurement state is **IDLE**. This applies to both temperature and battery measurements to prevent ADC resource conflicts.

Periodic Temperature Measurement The Sensors module supports automatic periodic temperature measurements that can be requested from two sources:

1. HOST Requests Applications can request periodic temperature measurements using:

```
SENSORS_TriggerPeriodicTemperatureMeasurement(uint32_t temperature_meas_interval_ms);
```

2. NBU Requests (wireless_mcu platforms only) The NBU (Narrow Band Unit) can request periodic temperature measurements. For more details about NBU temperature requests check NBU Integration Features section below.

Periodic Measurement Behavior Immediate Trigger: Every periodic request (HOST or NBU) triggers an **immediate measurement** before starting the interval timer.

Interval Management:

- **Minimum Interval Selection:** When both HOST and NBU request periodic measurements, the system uses the **shorter interval**
- **Dynamic Updates:** Changing the interval from either source immediately triggers a new measurement and restarts the timer with the new effective interval
- **Independent Control:** HOST and NBU can independently start/stop their periodic requests

One-shot Mode: Setting interval to 0 triggers a single measurement and stops periodic operation for that source.

Usage Examples

```
// HOST requests periodic measurement every 100ms
SENSORS_TriggerPeriodicTemperatureMeasurement(100);

// HOST requests one-shot measurement and stops periodic operation
SENSORS_TriggerPeriodicTemperatureMeasurement(0);
```

Concurrent Requests Behavior

HOST Interval	NBU Interval	Effective Interval	Notes
200ms	100ms	100ms	NBU interval is shorter
100ms	300ms	100ms	HOST interval is shorter
0 (stopped)	150ms	150ms	Only NBU active
200ms	0 (stopped)	200ms	Only HOST active
0 (stopped)	0 (stopped)	No periodic	Both stopped

Measurement State Management The Sensors module uses a state machine to manage ADC resource access:

States:

- **MEASUREMENT_IDLE:** No measurement in progress, ready to accept new requests
- **TMP_MEASUREMENT_ONGOING:** Temperature measurement in progress
- **BAT_MEASUREMENT_ONGOING:** Battery measurement in progress

Behavior:

- **Request Acceptance:** New measurement requests are only processed when state is MEASUREMENT_IDLE
- **Request Dropping:** If a measurement request is made while another measurement is ongoing, the new request is silently dropped
- **State Transitions:** State changes from IDLE to ONGOING when measurement starts, and back to IDLE when measurement completes
- **Recovery Mechanism:** For periodic measurements, if a trigger fails due to ongoing measurement, a 1ms retry timer is started

NBU Integration Features

Platform Support: NBU integration features are only supported on **wireless_mcu** platforms. These features are **not available** on RT, mcxw23, and rw61x platforms.

Periodic Temperature Measurement from NBU The Sensors module supports automatic periodic temperature measurements triggered by NBU requests. This feature allows the system to automatically measure temperature at regular intervals without manual intervention. This temperature data is used for XTAL32M calibration and Channel Sounding calibration.

How it works:

1. **NBU Request:** The NBU can request periodic temperature measurements by calling the registered callback with a measurement interval
2. **Timer-based Scheduling:** A timer manager is used to schedule periodic measurements based on the requested interval
3. **Workqueue Processing:** Temperature measurement triggers are processed through the system workqueue to avoid blocking operations
4. **Automatic Refresh:** Once a measurement is complete, the next measurement is automatically scheduled

Key Components:

- **Temperature Request Callback:** `Sensors_TemperatureReqCb()` - Handles NBU requests for periodic measurements
- **Timer Callback:** `Sensors_TempMeasTimerCallback()` - Triggers measurements on timer expiration
- **Work Handler:** `Sensors_PeriodicTempWorkHandler()` - Processes temperature measurement requests in workqueue context
- **Ready Callback:** `Sensors_TemperatureReadyCb()` - Called when temperature measurement is complete

Usage:

- **One-shot measurement:** NBU requests with interval = 0
- **Periodic measurement:** NBU requests with interval > 0 (in milliseconds)
- The periodic measurement continues until a new request changes the interval or stops it

Temperature Filtering and NBU Communication

Modified Moving Average Filter On wireless_mcu platforms, temperature readings are processed through a Modified Moving Average (MMA) filter to reduce noise and prevent unnecessary NBU notifications. This filter helps to:

- **Limit XTAL32M calibration changes:** Prevents excessive calibration adjustments that could degrade RF performances
- **Optimize power consumption:** Reduces NBU wake-up events by limiting notifications to significant temperature changes
- **Reduce noise:** Smooths out temperature reading fluctuations for more stable system behavior

The XTAL32M calibration process uses a LUT (Look-Up Table) that contains pre-characterized calibration values mapping temperature ranges to optimal XTAL32M frequency adjustments, enabling precise crystal oscillator tuning based on thermal conditions.

The LUT is registered using `PLATFORM_RegisterXtal32MTempCompLut()` and this is currently done in `examples/_common/project_segments/wireless/wireless_mcu/app_common/hardware_init.c` and requires `gBoardUseXtal32MTempComp` to be enabled.

Configuration: Filter size is controlled by `PLATFORM_TEMPERATURE_FILTER_SIZE` (default: 4 samples)

NBU Temperature Threshold Temperature values are only sent to the NBU when they exceed a configurable threshold:

Threshold is Controlled by `PLATFORM_TEMP_SENT_NBU_THRESHOLD` (default: 10 tenths of degrees Celsius = 1°C)

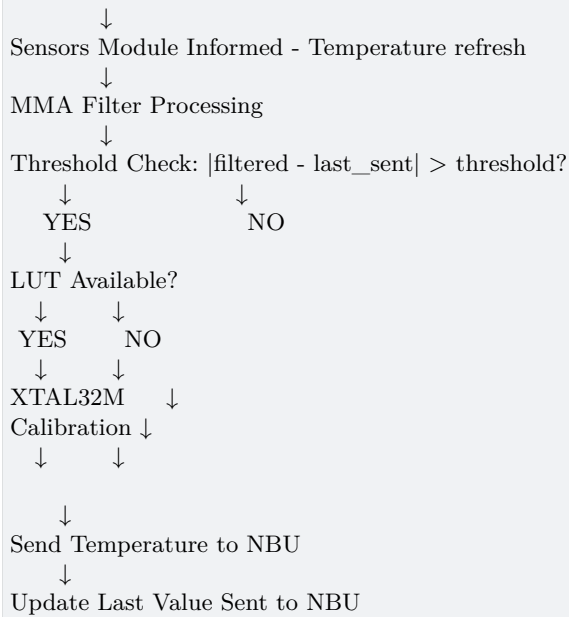
Temperature Processing Flow:

1. **Raw Measurement:** ADC interrupt captures temperature reading
2. **Sensors Module Informed** Sensors refresh the temperature value
3. **Filtering:** Raw value is processed through MMA filter
4. **Threshold Check:** Filtered value is compared against last sent value $\pm$ threshold
5. **NBU Communication:** If threshold is exceeded:
 - XTAL32M calibration is performed (if LUT is available)
 - Temperature value is sent to NBU
 - Last sent value is updated

```
Temperature Request
↓
ADC Interrupt - Measurement is ready
↓
Raw Temperature Reading
```

(continues on next page)

(continued from previous page)

**Platform Configuration Macros:**

```
// Temperature filter size (number of samples for averaging)
#define PLATFORM_TEMPERATURE_FILTER_SIZE 4U

// Threshold for sending temperature to NBU (in tenths of degrees Celsius)
#define PLATFORM_TEMP_SENT_NBU_THRESHOLD 10 // 1°C
```

API Usage Examples**Manual Temperature Measurement**

```
int32_t temperature1, temperature2;
SENSORS_TriggerTemperatureMeasurement();
...
temperature1 = (SENSORS_RefreshTemperatureValue())/10;
temperature2 = (SENSORS_GetTemperature())/10;
```

In this case temperature1 is equal to temperature2, both are in degree celsius.

Battery Level Measurement

```
uint8_t batteryLevel;
SENSORS_TriggerBatteryMeasurement();
...
batteryLevel = SENSORS_RefreshBatteryLevel();
// batteryLevel is now in percentage (0-100%)
```

Handling Concurrent Requests

```
// First request - will be accepted
SENSORS_TriggerTemperatureMeasurement();

// Second request while first is ongoing - will be dropped silently
```

(continues on next page)

(continued from previous page)

```

SENSORS_TriggerBatteryMeasurement(); // This will be ignored

// Wait for first measurement to complete
int32_t temp = SENSORS_RefreshTemperatureValue();

// Now battery measurement can be triggered
SENSORS_TriggerBatteryMeasurement(); // This will be accepted

```

Low Power Optimization - Wake-up Temperature Measurement

```

// In low power exit callback (interrupts masked, scheduler disabled)
/* Trig temperature measurement on wake-up.
 * If a temperature change is detected,
 * the new value will be sent to the NBU for appropriate XTAL32M trimming.
 * During channel sounding measurements, this temperature is also used for RTT compensation.
 * Since this function runs with interrupts masked and the scheduler disabled,
 * the unsafe variant must be used to avoid mutex acquisition.
 */
SENSORS_TriggerTemperatureMeasurementUnsafe();

```

This approach optimizes power consumption and provides additional benefits:

- **XTAL32M Trimming:** Temperature changes are sent to NBU for crystal oscillator calibration only if they exceed the configured threshold after filtering
- **RTT Compensation:** This temperature data is used by the NBU for RTT compensation during channel sounding measurements
- **Safe Execution:** Uses the unsafe variant to avoid mutex acquisition in interrupt context with scheduler disabled In case of periodic measurement:
- **Power Optimization:** Triggers temperature measurements when the system is already awake, avoiding dedicated timer wake-ups
- **Timer Reset:** Resets the periodic measurement timer to prevent unnecessary wake-ups

Note: This optimization is implemented by default in wireless platform examples when `gAppUseSensors_d` is enabled. The temperature measurement is automatically triggered in the low power exit callback (`BOARD_ExitLowPowerCb()`) in `examples/_common/project_segments/wireless/wireless_mcu/board_lp.c`.

Constant macro definitions Name :

```

#define VALUE_NOT_AVAILABLE_8 0xFFu
#define VALUE_NOT_AVAILABLE_32 0xFFFFFFFFu

```

Description :

Defines the error value that can be compared to the value obtain on the ADC.

SFC : Smart Frequency Calibration

Overview The Smart Frequency Calibration module provides operations and calibration for the FRO32K source clock. This module is split between main core and Radio core:

- `fwk_rf_sfc.[ch]`: RF_SFC module on Radio core that provides Main FRO32K measurement/calibration and state machine in synchronizaton with Radio domain activities. See details below.

- `fwk_sfc.h`: SFC module on host core that provides type definition for usage with `fwk_platform_ics.[ch]` with `PLATFORM_FwkSrvSetRfSfcConfig()` API and `fwk_platform_ble.c` for received callback from the NBU core

Host SFC Module

Algorithm parametrization This module provides ability to configure the RF_SFC module by sending message to Radio core through `fwk_platform_ics.c` `PLATFORM_FwkSrvSetRfSfcConfig()`:

- Filter size
- Maximum ppm threshold
- Maximum calibration interval
- Number of sample in filter to switch from convergence to monitor mode

Ppm target The ppm target is the deviation from the target clock accepted by the algorithm. When the deviation is larger than the ppm target. The algorithm will update the trimming value and reset the filter. The ppm target cannot be more aggressive `RF_SFC_MAXIMAL_PPM_TARGET` in order to avoid having to update trimming value at each measurement.

Filter size Filter size must be included between `RF_SFC_MINIMAL_FILTER_SIZE` and `RF_SFC_MAXIMAL_FILTER_SIZE`. See *Filtering and Frequency estimation* section for more details on the parameter.

Maximum calibration interval In monitor mode, new measurement are triggered by low-power entry/exit. If the NBU core has a lot of radio activity it could never enter lowpower. The maximum calibration interval is here to ensure a measurement is done regularly. When executing idle the SFC module checks when the last measurement has been done, if it has been too long, it reset the filter and forces a new measurement

Trig sample number The trig sample number is the number of samples needed by the algorithm in its filter to switch from convergence to monitor mode. Having more than one sample in convergence mode allows to confirm the trimming value that we have set.

SFC debug information On the other way, the RF_SFC from Radio core sends back notifications to SFC module on main core using RX callback `PLATFORM_RegisterFroNotificationCallback()` from `fwk_platform_ics.h` and such information:

- last measured frequency
- average ppm from 32768Khz frequency
- last ppm measured from 32768Khz frequency
- FRO trimming value

RF_SFC module The RF_SFC module provides the functionality to calibrate the FRO32K source clock during Initialization and radio activity.

The RF_SFC is mostly used on XTAL32K less solution when no 32Khz crystal is soldered on the board. It allows to calibrate the FRO32K source clock to the desired frequency to keep Radio time base within the allowed tolerance given by the connectivity standards. However, even on a XTAL32K solution, the RF_SFC is also used during Initialization until the XTAL32K is up and

running in the system. The system firstly runs on the FRO32K clock source then switch to the XTAL32K clock source when it is ready with enough accuracy. This allows to save significant boot time as the FRO32K start up (including calibration) is much faster compared to XTAL32K .

This module will handle:

- FRO32K clock frequency measurement against 32Mhz crystal. It schedules appropriately the start of the measurement and gets the result when completed,
- Filter and estimate the 32Khz frequency value and error by averaging from the last measurements,
- FRO32K calibration in order to update the trimming value to reduce the frequency error on the clock.

The targeted frequency offset shall be within 200ppm. The RF_SFC will handle two modes of operation:

- Convergence mode: when frequency estimation is above 200pm,
- Monitor mode: when frequency estimation is below 200pm.

The RF_SFC module works in active and all low power modes on NBU domain, or on host application domain except power down mode. Power down mode on host application domain is not supported with the FRO32K configuration as clock source.

Feature enablement Enabling the FRO32K is done by calling the PLATFORM_InitFro32K() function during application initialization in hardware_init.c file, in BOARD_InitHardware() function. If FRO32K is not enabled, Oscillator XTAL32K shall be called instead by calling PLATFORM_InitOsc32K() function. The call to PLATFORM_InitFro32K() from BOARD_InitHardware() can be done by setting the Compilation flag gBoardUseFro32k_d to 1 in hardware_init.c or any header files included from this file.

```
#define gBoardUseFro32k_d 1
```

Detailed description

Frequency measurements When NBU low power is enabled, the frequency measurements are triggered on Low power wake-up by HW signal. The SFC process called from Idle task will check regularly the completion of the frequency measurement. When the measurement is done, it goes to filtering and frequency estimation process. The frequency measurement duration depends on monitor mode or convergence mode: In convergence mode, the frequency measurement duration is 0.5ms while it is 2ms in monitor mode. In monitor mode, the duration value remains less than the minimal radio activity duration so it does not impact the low power consumption in monitoring mode.

Filtering and Frequency estimation The FRO32KHz frequency measurement values are noisy because of thermal noise on the FRO32K itself. Also, the frequency measurement can introduce some error. In monitoring mode, it is required to filter the measurements by applying an exponential filter: $\text{new\_estimation} = (\text{new\_measurement} + ((1 \ll n) - 1) * \text{last\_estimation}) \gg n$

Default value for n is 7 (meaning 128 samples in the averaging window).

Frequency calibration When the frequency estimation gets higher than the targeted 200ppm target, the RF_SFC updates the trimming value for one positive or negative increment. For this purpose, it requires to:

- wake up the host application domain and keep the domain active,

- update the trim register of the FRO32K, this register is used to trim the capacitance value of the FRO32K,
- re-allow the host application domain to enter low power.

A slight power impact is expected during a calibration update due to host domain wake-up.

Operational modes When the low power mode is enabled on NBU power domain, RF SFC handles two modes of operation: convergence and monitor modes. However, when low power is disabled on NBU power domain, only convergence mode is supported.

Convergence mode Convergence mode is used when the estimated FRO32K frequency is above 200ppm or when the filter has been reset. Typically this occurs :

- During Power ON reset or other reset when NBU is switched OFF
- When temperature varies and FRO32K frequency deviates outside 200ppm threshold target
- When no calibration has been done during some time as we discard old values that could influence the algorithm

The convergence mode process typically starts with a FRO32K trim register update, performs a frequency measurement and the FRO32K trim register is updated until the measured frequency gets below 200ppm. These operations are repeated in a loop until the estimated frequency value gets below 200ppm. When below 200ppm during multiple measurements, the RC SFC switches to Monitoring mode. The convergence mode is only a transition mode to monitoring mode. In convergence mode, the NBU power domain does not go to low power. The convergence mode time duration depends on the initial frequency error of the FRO32K. Default frequency measurement duration is 0.5ms so 20 measurements (given as example only) will require less than 10 ms to converge.

Monitoring mode Monitoring mode is used when the estimated FRO32K frequency is below 200ppm. In this mode, the measurement is triggered on NBU domain wake up from low power mode using an internal hardware signal. The exponential filter is applied to compute the frequency estimation. If the frequency estimation value is still within 200ppm, the NBU power domain is allowed to go to low power. If the estimated value gets above the 200ppm threshold, the RF SFC switch back to convergence mode. The trim register is updated by one increment (positive or negative) and because the frequency has been adjusted and changed, the estimated filtered frequency is reset to discard all previous measurements. Going back to convergence mode typically happens during a temperature gradient. If the temperature is constant, it is not expected to have the estimated value to go beyond 200ppm so no calibration should be required.

Initialization and configuration During initialization, the RF SFC module will block the Radio Software until monitoring mode is reached. This is to prevent the radio from running with an inaccurate time base due to an important 32k clock frequency error.

Initialization and configuration is done by the NBU core. The configuration parameters can set up:

- The 200ppm target threshold. This value shall be 200ppm or higher.
- The filtering number n (see section above), It shall be between 0 and 8. Default is 7 which is similar to an averaging filter of 128 samples. A higher value will be more robust against noise. A lower value will track temperature variation more faster.

In order to prevent the host application domain from going into power down mode (power down mode not supported with FRO32K as clock source), the fwkSrvLowPowerConstraintCallbacks functions structure is registered to the Framework service on host application core from fwk_platform_lowpower.c file, PLATFORM_LowPowerInit() function. The NBU code applies a

low power Deep Sleep constraint to the application core. This constraint is released when the NBU firmware has no activity to do and re-applied when a new activity starts.

Lowpower impact

Power impact during active mode: In monitoring mode (this should be 99.9% of the time if temperature does not vary), the FRO32KHz frequency measurements are performed during a Radio activity so it does not increase the active current as the sources clocks are already active. Also, it does not increase the active time as the measurement takes less time than an advertising event or connection event so no impact on power consumption.

The main power impact will be in convergence mode. In this case, measurements/calibrations are done in loop until the monitoring mode is reached (frequency error less than 200ppm). This could happen:

- During power ON reset,
- When temperature varies: The frequency will deviate from 32768Hz and FRO32K trimming register correction will need to be updated for that,
- When no measurement has been done during some time as we cannot predict if the FRO has drifted, so we discard older values and start convergence mode.

When FRO32K frequency needs to be adjusted, the NBU core will wake-up the main power domain and will update the FRO32K trimming register.

Power impact during low power mode: The power consumption in low power mode will increase slightly due to running FRO32K compared to XTAL32K. The power consumption of FRO32K typically consumes 350nA while it is only 100nA with XTAL32K. Refer to the product datasheet for the exact numbers.

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