



MCUXpresso SDK Documentation

Release 25.09.00-pvw2



NXP
Aug 12, 2025



Table of contents

1	MCXW23-EVK	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK GitHub	3
1.2.1	Getting Started with MCUXpresso SDK Repository	4
1.2.2	How to determine COM Port	16
1.2.3	Updating debugger firmware	17
1.3	Release Notes	18
1.3.1	MCUXpresso SDK Release Notes	18
1.4	ChangeLog	24
1.4.1	MCUXpresso SDK Changelog	24
1.5	Driver API Reference Manual	68
1.6	Middleware Documentation	68
1.6.1	Wireless Bluetooth LE host stack and applications	68
1.6.2	Wireless Connectivity Framework	68
1.6.3	FreeRTOS	131
2	MCXW236B	133
2.1	ANACTRL: Analog Control Driver	133
2.2	CASPER: The Cryptographic Accelerator and Signal Processing Engine with RAM sharing	138
2.3	casper_driver	138
2.4	casper_driver_pkha	141
2.5	CDOG	144
2.6	CRC: Cyclic Redundancy Check Driver	148
2.7	CTIMER: Standard counter/timers	151
2.8	DMA: Direct Memory Access Controller Driver	160
2.9	FLEXCOMM: FLEXCOMM Driver	177
2.10	FLEXCOMM Driver	177
2.11	GINT: Group GPIO Input Interrupt Driver	178
2.12	Hashcrypt: The Cryptographic Accelerator	181
2.13	Hashcrypt Background HASH	181
2.14	Hashcrypt common functions	182
2.15	Hashcrypt AES	184
2.16	Hashcrypt HASH	189
2.17	I2C: Inter-Integrated Circuit Driver	190
2.18	I2C DMA Driver	190
2.19	I2C Driver	192
2.20	I2C Master Driver	196
2.21	I2C Slave Driver	205
2.22	INPUTMUX: Input Multiplexing Driver	214
2.23	Common Driver	215
2.24	GPIO: General Purpose I/O	227
2.25	IOCON: I/O pin configuration	229
2.26	Mailbox	230
2.27	MRT: Multi-Rate Timer	231
2.28	OSTIMER: OS Event Timer Driver	236

2.29	PINT: Pin Interrupt and Pattern Match Driver	239
2.30	PLU: Programmable Logic Unit	248
2.31	PUF: Physical Unclonable Function	257
2.32	RTC: Real Time Clock	259
2.33	SCTimer: SCTimer/PWM (SCT)	265
2.34	SPI: Serial Peripheral Interface Driver	282
2.35	SPI DMA Driver	282
2.36	SPI Driver	286
2.37	SPIFI: SPIFI flash interface driver	294
2.38	SPIFI DMA Driver	303
2.39	SPIFI Driver	303
2.40	TRNG: True Random Number Generator	303
2.41	USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver	307
2.42	USART DMA Driver	307
2.43	USART Driver	310
2.44	UTICK: MicroTick Timer Driver	326
2.45	WWDT: Windowed Watchdog Timer Driver	327
3	Middleware	331
3.1	Wireless	331
3.1.1	NXP Wireless Framework and Stacks	331
4	RTOS	333
4.1	FreeRTOS	333
4.1.1	FreeRTOS kernel	333
4.1.2	FreeRTOS drivers	339
4.1.3	backoffalgorithm	339
4.1.4	corehttp	342
4.1.5	corejson	344
4.1.6	coremqtt	347
4.1.7	coremqtt-agent	350
4.1.8	corepkcs11	354
4.1.9	freertos-plus-tcp	357

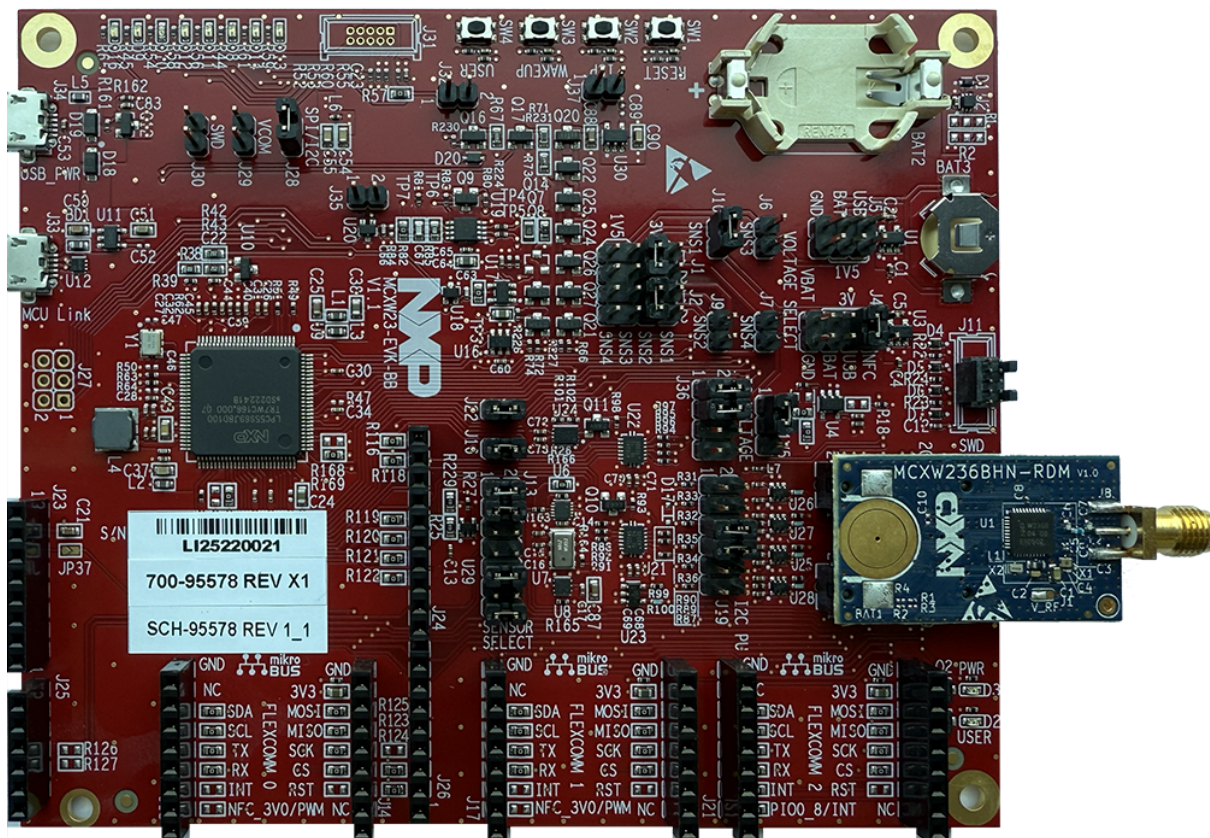
This documentation contains information specific to the mcxw23evk board.

Chapter 1

MCXW23-EVK

1.1 Overview

The NXP MCXW23-EVK is a development board for the MCXW23 32 MHz Arm Cortex-M33 microcontroller.



MCU device and part on board is shown below:

- Device: MCXW236B
- PartNumber: MCXW236BIHNAR

1.2 Getting Started with MCUXpresso SDK GitHub

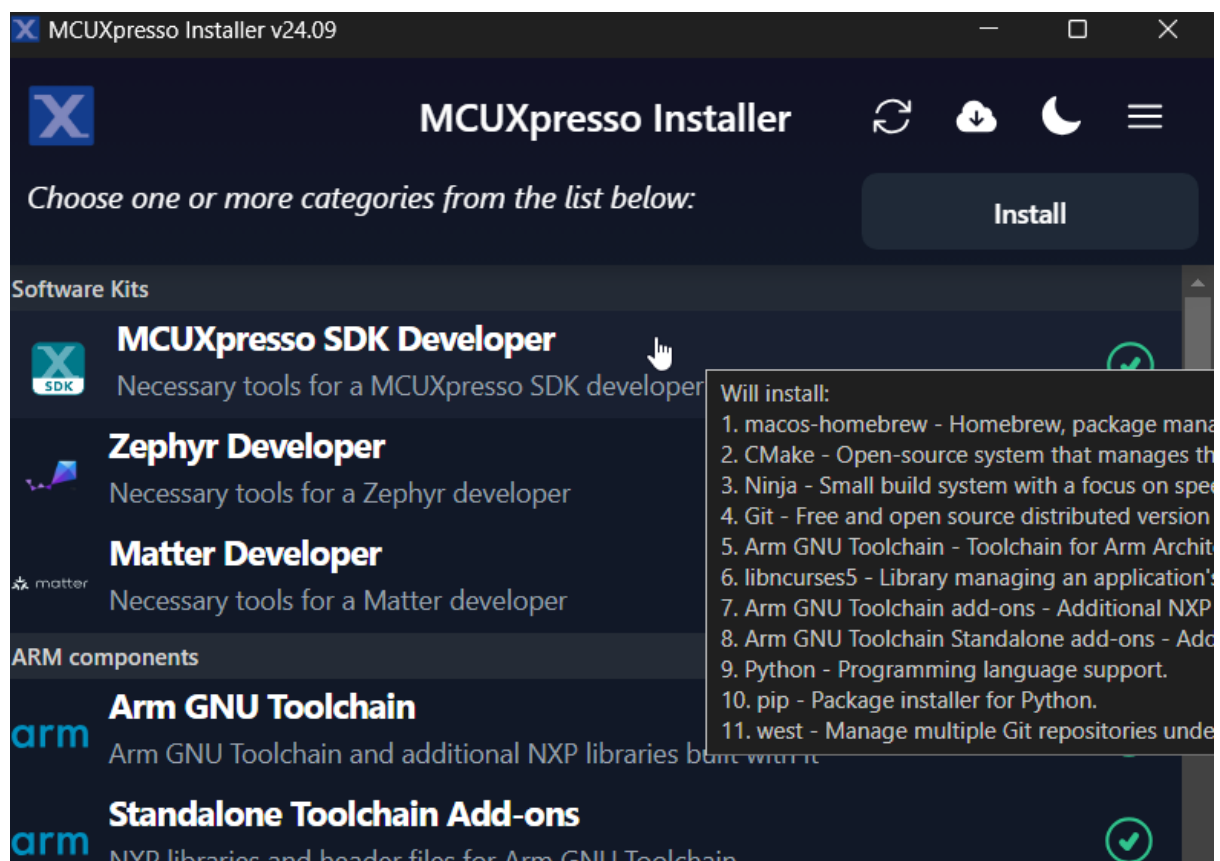
1.2.1 Getting Started with MCUXpresso SDK Repository

Installation

NOTE

If the installation instruction asks/selects whether to have the tool installation path added to the PATH variable, agree/select the choice. This option ensures that the tool can be used in any terminal in any path. [Verify the installation](#) after each tool installation.

Install Prerequisites with MCUXpresso Installer The MCUXpresso Installer offers a quick and easy way to install the basic tools needed. The MCUXpresso Installer can be obtained from <https://github.com/nxp-mcuxpresso/vscode-for-mcux/wiki/Dependency-Installation>. The MCUXpresso Installer is an automated installation process, simply select MCUXpresso SDK Developer from the menu and click install. If you prefer to install the basic tools manually, refer to the next section.



Alternative: Manual Installation

Basic tools

Git Git is a free and open source distributed version control system. Git is designed to handle everything from small to large projects with speed and efficiency. To install Git, visit the official [Git website](#). Download the appropriate version (you may use the latest one) for your operating system (Windows, macOS, Linux). Then run the installer and follow the installation instructions.

User `git --version` to check the version if you have a version installed.

Then configure your username and email using the commands:

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

Python Install python 3.10 or latest. Follow the [Python Download](#) guide.

Use `python --version` to check the version if you have a version installed.

West Please use the west version equal or greater than 1.2.0

```
# Note: you can add option '--default-timeout=1000' if you meet connection issue. Or you may set a different
↪source using option '-i'.
# for example, in China you could try: pip install -U west -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install -U west
```

Build And Configuration System

CMake It is strongly recommended to use CMake version equal or later than 3.30.0. You can get latest CMake distributions from [the official CMake download page](#).

For Windows, you can directly use the .msi installer like `cmake-3.31.4-windows-x86_64.msi` to install.

For Linux, CMake can be installed using the system package manager or by getting binaries from [the official CMake download page](#).

After installation, you can use `cmake --version` to check the version.

Ninja Please use the ninja version equal or later than 1.12.1.

By default, Windows comes with the Ninja program. If the default Ninja version is too old, you can directly download the [ninja binary](#) and register the ninja executor location path into your system path variable to work.

For Linux, you can use your [system package manager](#) or you can directly download the [ninja binary](#) to work.

After installation, you can use `ninja --version` to check the version.

Kconfig MCUXpresso SDK uses Kconfig python implementation. We customize it based on our needs and integrate it into our build and configuration system. The Kconfiglib sources are placed under `mcuxsdk/scripts/kconfig` folder.

Please make sure [python](#) environment is setup ready then you can use the Kconfig.

Ruby Our build system supports IDE project generation for iar, mdk, codewarrior and xtensa to provide OOB from build to debug. This feature is implemented with ruby. You can follow the guide [ruby environment setup](#) to setup the ruby environment. Since we provide a built-in portable ruby, it is just a simple one cmd installation.

If you only work with CLI, you can skip this step.

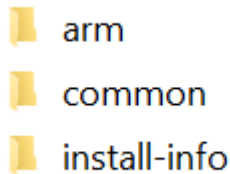
Toolchain MCUXpresso SDK supports all mainstream toolchains for embedded development. You can install your used or interested toolchains following the guides.

Toolchain	Download and Installation Guide	Note
Armgcc	Arm GNU Toolchain Install Guide	ARMGCC is default toolchain
IAR	IAR Installation and Licensing quick reference guide	
MDK	MDK Installation	
Armclang	Installing Arm Compiler for Embedded	
Zephyr	Zephyr SDK	
Codewarrior	NXP CodeWarrior	
Xtensa	Tensilica Tools	
NXP S32Compiler RISC-V Zen-V	NXP Website	

After you have installed the toolchains, register them in the system environment variables. This will allow the west build to recognize them:

Toolchain	Environment Variable	Example	Cmd Line Argument
Armgcc	AR-MGCC_DIR	C:\armgcc for windows/usr for Linux. Typically arm-none-eabi-* is installed under /usr/bin	– toolchain armgcc
IAR	IAR_DIR	C:\iar\ewarm-9.60.3 for Windows/opt/iarsystems/bxarm-9.60.3 for Linux	– toolchain iar
MDK	MDK_DIR	C:\Keil_v5 for Windows.MDK IDE is not officially supported with Linux.	– toolchain mdk
Armclang	ARM-CLANG_DIR	C:\ArmCompilerforEmbedded6.22 for Windows/opt/ArmCompilerforEmbedded6.21 for Linux	– toolchain mdk
Zephyr	ZEPHYR_DIR	c:\NXP\zephyr-sdk-<version> for windows/opt/zephyr-sdk-<version> for Linux	– toolchain zephyr
CodeWarrior	CW_DIR	C:\Freescall\CW MCU v11.2 for windowsCodeWarrior is not supported with Linux	– toolchain code-warrior
Xtensa	XCC_DIR	C:\xtensa\XtDevTools\install\tools\RI-2023.11-win32\XtensaTools for windows/opt/xtensa/XtDevTools/install/tools/RI-2023.11-Linux/XtensaTools for Linux	– toolchain xtensa
NXP S32Compiler RISC-V Zen-V	RISCVL-LVM_DIR	C:\riscv-llvm-win32_b298_b298_2024.08.12 for Windows/opt/riscv-llvm-Linux-x64_b298_b298_2024.08.12 for Linux	– toolchain riscv-llvm

- The <toolchain>_DIR is the root installation folder, not the binary location folder. For IAR, it is directory containing following installation folders:



- MDK IDE using armclang toolchain only officially supports Windows. In Linux, please directly use armclang toolchain by setting ARMCLANG_DIR. In Windows, since most Keil users will install MDK IDE instead of standalone armclang toolchain, the MDK_DIR has higher priority than ARMCLANG_DIR.
- For Xtensa toolchain, please set the XTENSA_CORE environment variable. Here's an example list:

Device Core	XTENSA_CORE
RT500 fusion1	nxp_rt500_RI23_11_newlib
RT600 hifi4	nxp_rt600_RI23_11_newlib
RT700 hifi1	rt700_hifi1_RI23_11_nlib
RT700 hifi4	t700_hifi4_RI23_11_nlib
i.MX8ULP fusion1	fusion_nxp02_dsp_prod

- In Windows, the short path is used in environment variables. If any toolchain is using the long path, you can open a command window from the toolchain folder and use below command to get the short path: for %i in (.) do echo %~fsi

Tool installation check Once installed, open a terminal or command prompt and type the associated command to verify the installation.

If you see the version number, you have successfully installed the tool. Else, check whether the tool's installation path is added into the PATH variable. You can add the installation path to the PATH with the commands below:

- Windows: Open command prompt or powershell, run below command to show the user PATH variable.

```
reg query HKEY_CURRENT_USER\Environment /v PATH
```

The tool installation path should be C:\Users\xxx\AppData\Local\Programs\Git\cmd. If the path is not seen in the output from above, append the path value to the PATH variable with the command below:

```
reg add HKEY_CURRENT_USER\Environment /v PATH /d "%PATH%;C:\Users\xxx\AppData\Local\Programs\Git\cmd"
```

Then close the command prompt or powershell and verify the tool command again.

- Linux:
 1. Open the \$HOME/.bashrc file using a text editor, such as vim.
 2. Go to the end of the file.
 3. Add the line which appends the tool installation path to the PATH variable and export PATH at the end of the file. For example, export PATH="/Directory1:\$PATH".
 4. Save and exit.

5. Execute the script with `source .bashrc` or reboot the system to make the changes live. To verify the changes, run `echo $PATH`.
- macOS:
 1. Open the `$HOME/.bash_profile` file using a text editor, such as `nano`.
 2. Go to the end of the file.
 3. Add the line which appends the tool installation path to the `PATH` variable and export `PATH` at the end of the file. For example, export `PATH="/Directory1:$PATH"`.
 4. Save and exit.
 5. Execute the script with `source .bash_profile` or reboot the system to make the changes live. To verify the changes, run `echo $PATH`.

Get MCUXpresso SDK Repo

Establish SDK Workspace To get the MCUXpresso SDK repository, use the `west` tool to clone the manifest repository and checkout all the west projects.

```
# Initialize west with the manifest repository
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests/ mcuxpresso-sdk

# Update the west projects
cd mcuxpresso-sdk
west update

# Allow the usage of west extensions provided by MCUXpresso SDK
west config commands.allow__extensions true
```

Install Python Dependency(If do tool installation manually) To create a Python virtual environment in the west workspace core repo directory `mcuxsdk`, follow these steps:

1. Navigate to the core directory:

```
cd mcuxsdk
```

2. [Optional] Create and activate the virtual environment: If you don't want to use the python virtual environment, skip this step. **We strongly suggest you use `venv` to avoid conflicts with other projects using python.**

```
python -m venv .venv

# For Linux/MacOS
source .venv/bin/activate

# For Windows
.\.venv\Scripts\activate
# If you are using powershell and see the issue that the activate script cannot be run.
# You may fix the issue by opening the powershell as administrator and run below command:
powershell Set-ExecutionPolicy RemoteSigned
# then run above activate command again.
```

Once activated, your shell will be prefixed with `(.venv)`. The virtual environment can be deactivated at any time by running `deactivate` command.

Remember to activate the virtual environment every time you start working in this directory. If you are using some modern shell like `zsh`, there are some powerful plugins to help you auto switch `venv` among workspaces. For example, [zsh-autoswitch-virtualenv](#).

3. Install the required Python packages:

```
# Note: you can add option '--default-timeout=1000' if you meet connection issue. Or you may set a ↵
↵different source using option '-i'.
# for example, in China you could try: pip3 install -r mcuxsdk/scripts/requirements.txt -i https://pypi.
↵tuna.tsinghua.edu.cn/simple
pip install -r scripts/requirements.txt
```

Explore Contents

This section helps you build basic understanding of current fundamental project content and guides you how to build and run the provided example project in whole SDK delivery.

Folder View The whole MCUXpresso SDK project, after you have done the west init and west update operations follow the guideline at [Getting Started Guide](#), have below folder structure:

Folder	Description
mani-fests	Manifest repo, contains the manifest file to initialize and update the west workspace.
mcuxsdk	The MCUXpresso SDK source code, examples, middleware integration and script files.

All the projects record in the [Manifest repo](#) are checked out to the folder mcuxsdk/, the layout of mcuxsdk folder is shown as below:

Folder	Description
arch	Arch related files such as ARM CMSIS core files, RISC-V files and the build files related to the architecture.
cmake	The cmake modules, files which organize the build system.
com- po- nents	Software components.
de- vices	Device support package which categorized by device series. For each device, header file, feature file, startup file and linker files are provided, also device specific drivers are included.
docs	Documentation source and build configuration for this sphinx built online documentation.
drivers	Peripheral drivers.
ex- am- ples	Various demos and examples, support files on different supported boards. For each board support, there are board configuration files.
mid- dle- ware	Middleware components integrated into SDK.
rtos	Rtos components integrated into SDK.
scripts	Script files for the west extension command and build system support.
svd	Svd files for devices, this is optional because of large size. Customers run west manifest config group.filter +optional and west update mcux-soc-svd to get this folder.

Examples Project The examples project is part of the whole SDK delivery, and locates in the folder mcuxsdk/examples of west workspace.

Examples files are placed in folder of <example_category>, these examples include (but are not limited to)

- `demo_apps`: Basic demo set to start using SDK, including `hello_world` and `led_blinky`.
- `driver_examples`: Simple applications that show how to use the peripheral drivers for a single use case. These applications typically only use a single peripheral but there are cases where multiple peripherals are used (for example, SPI transfer using DMA).

Board porting layers are placed in folder of `_boards/<board_name>` which aims at providing the board specific parts for examples code mentioned above.

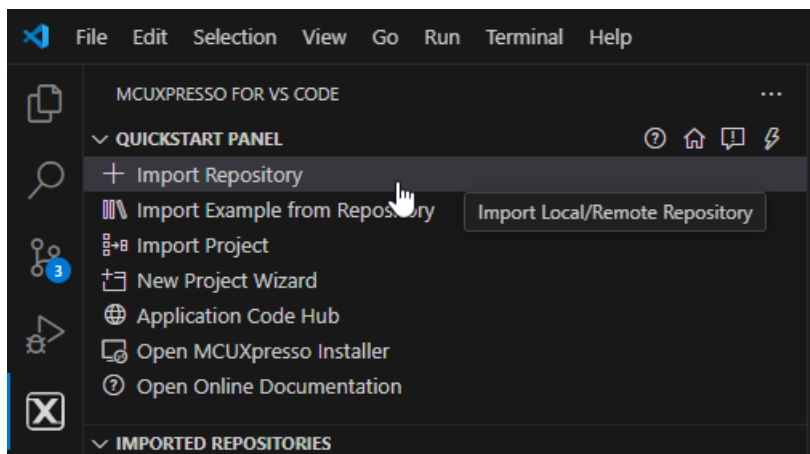
Run a demo using MCUXpresso for VS Code

This section explains how to configure MCUXpresso for VS Code to build, run, and debug example applications. This guide uses the `hello_world` demo application as an example. However, these steps can be applied to any example application in the MCUXpresso SDK.

Build an example application This section assumes that the user has already obtained the SDK as outlined in [Get MCUXpresso SDK Repo](#).

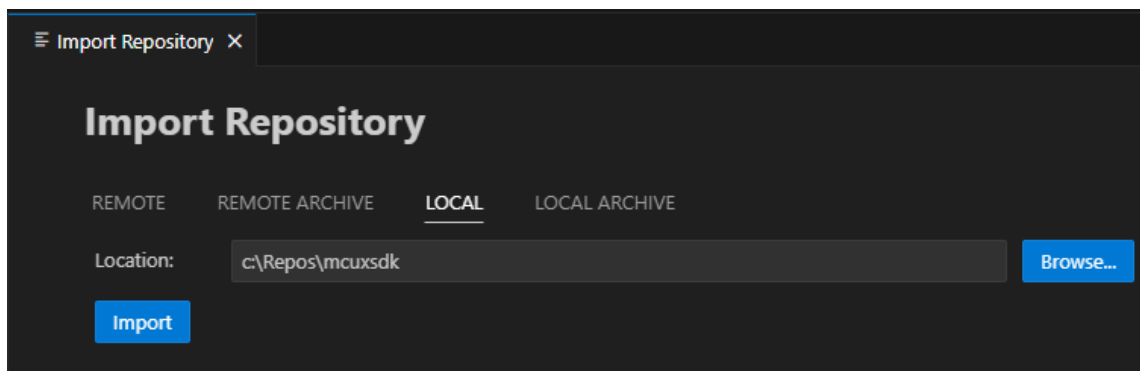
To build an example application:

1. Import the SDK into your workspace. Click **Import Repository** from the **QUICKSTART PANEL**.

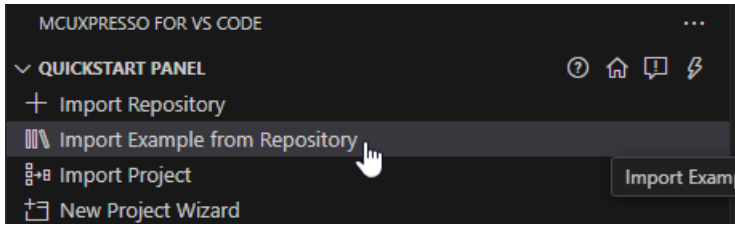


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

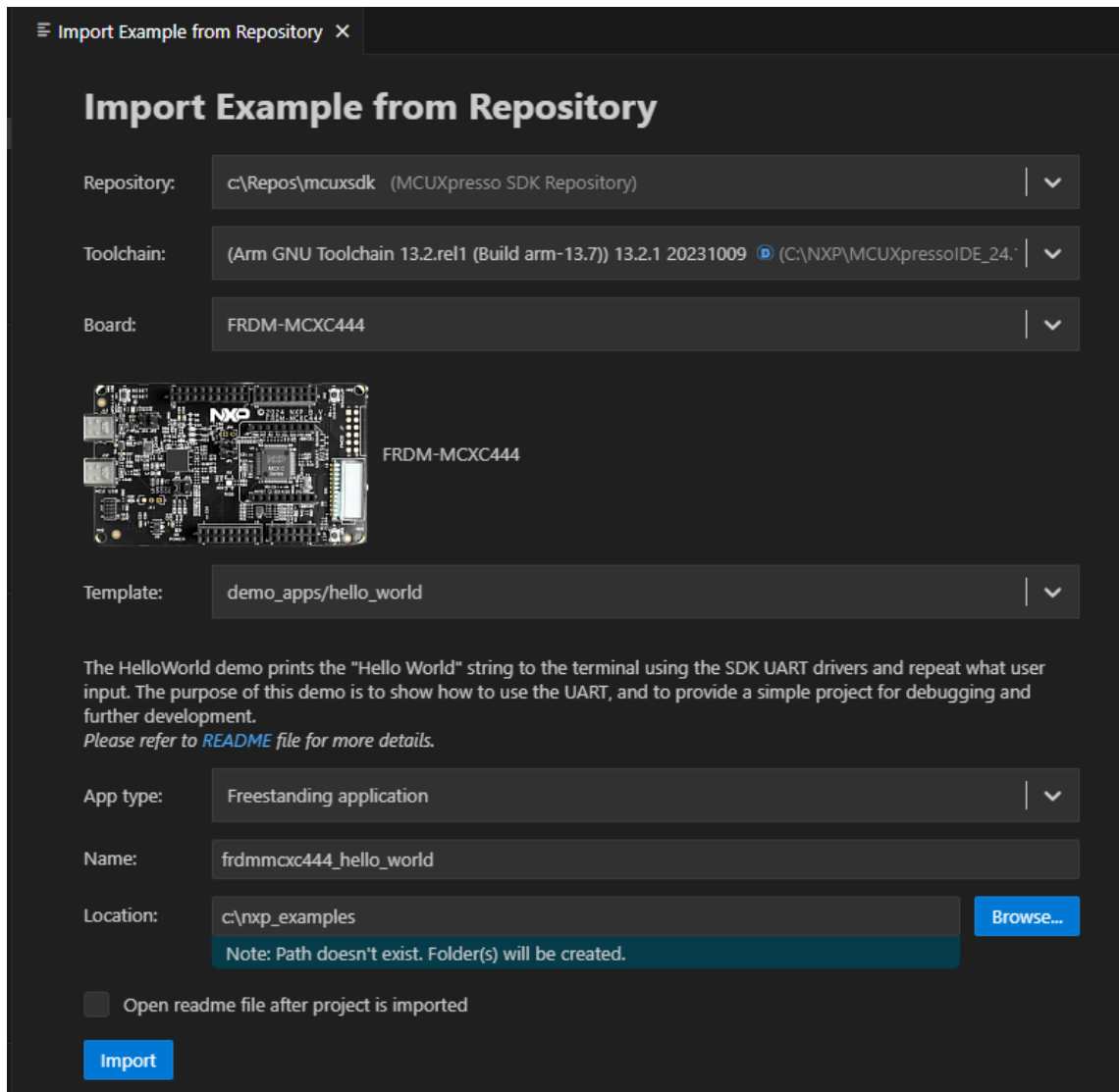
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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

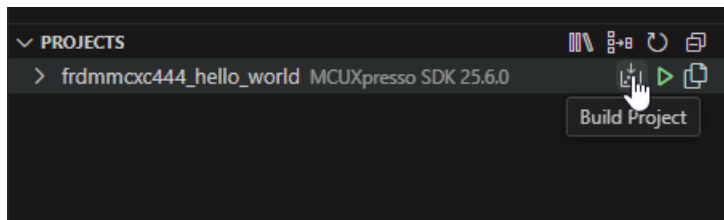


In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.

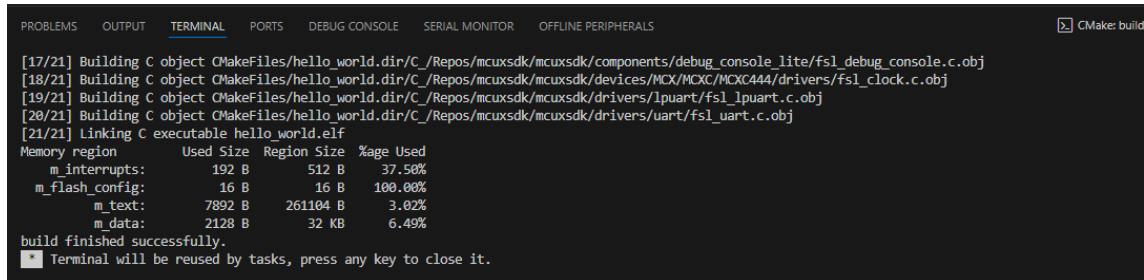


Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Free-standing applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Free-standing examples can be imported to a user-defined location. Select between these by designating your selection in the **App type** dropdown menu.

3. VS Code will prompt you to confirm if the imported files are trusted. Click **Yes**.
4. Navigate to the **PROJECTS** view. Find your project and click the **Build Project** icon.

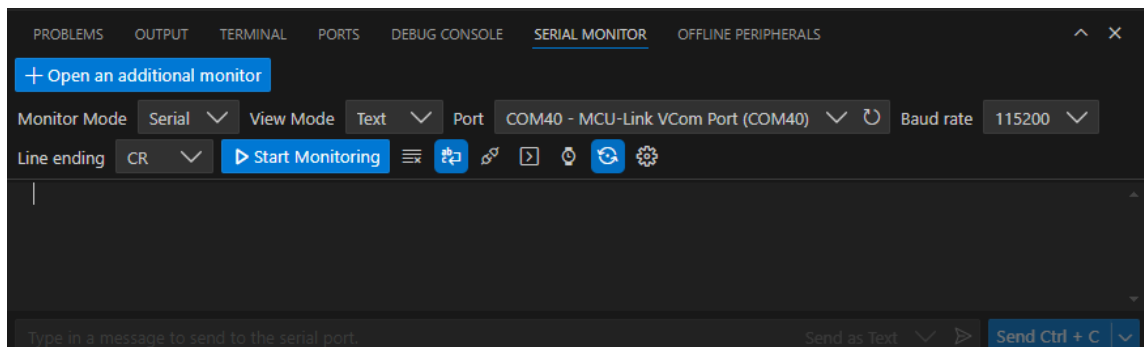


The integrated terminal will open at the bottom and will display the build output.

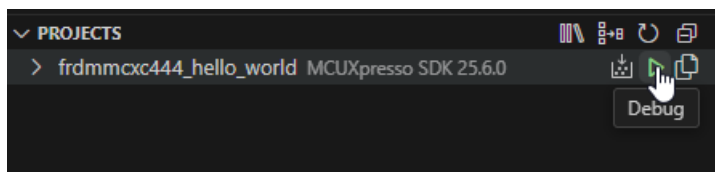


Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

1. Open the **Serial Monitor** from the VS Code's integrated terminal. Select the VCom Port for your device and set the baud rate to 115200.



2. Navigate to the **PROJECTS** view and click the play button to initiate a debug session.



The debug session will begin. The debug controls are initially at the top.


```

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

```

3. Click **Continue** on the debug controls to resume execution of the code. Observe the output on the **Serial Monitor**.

```

PROBLEMS  OUTPUT  TERMINAL  PERIPHERALS  RTOS DETAILS  PORTS  DEBUG CONSOLE  SERIAL MONITOR
+ Open an additional monitor
Monitor Mode Serial View Mode Text Port COM40 - MCU-Link VCom Port (COM40)
[Stop Monitoring] [Icons]
---- Opened the serial port COM40 ----
hello world.
|

```

Running a demo using ARMGCC CLI/IAR/MDK

Supported Boards Use the west extension `west list_project` to understand the board support scope for a specified example. All supported build command will be listed in output:

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

```
INFO: [ 1][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↳ evk9mimx8ulp -Dcore_id=cm33]
```

```
INFO: [ 2][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↳ evkbimxrt1050]
```

```
INFO: [ 3][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
```

(continues on next page)

(continued from previous page)

```

↪ evkbnimxrt1060]
INFO: [ 4][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1170 -Dcore_id=cm4]
INFO: [ 5][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1170 -Dcore_id=cm7]
INFO: [ 6][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1060]
INFO: [ 7][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt7ulp]
...

```

The supported toolchains and build targets for an example are decided by the example-self example.yml and board example.yml, please refer Example Toolchains and Targets for more details.

Build the project Use west build -h to see help information for west build command. Compared to zephyr's west build, MCUXpresso SDK's west build command provides following additional options for mcux examples:

- --toolchain: specify the toolchain for this build, default armgcc.
- --config: value for CMAKE_BUILD_TYPE. If not provided, build system will get all the example supported build targets and use the first debug target as the default one. Please refer Example Toolchains and Targets for more details about example supported build targets.

Here are some typical usages for generating a SDK example:

```

# Generate example with default settings, default used device is the mainset MK22F51212
west build -b frdmk22f examples/demo_apps/hello_world

# Just print cmake commands, do not execute it
west build -b frdmk22f examples/demo_apps/hello_world --dry-run

# Generate example with other toolchain like iar, default armgcc
west build -b frdmk22f examples/demo_apps/hello_world --toolchain iar

# Generate example with other config type
west build -b frdmk22f examples/demo_apps/hello_world --config release

# Generate example with other devices with --device
west build -b frdmk22f examples/demo_apps/hello_world --device MK22F12810 --config release

```

For multicore devices, you shall specify the corresponding core id by passing the command line argument -Dcore_id. For example

```

west build -b evkbnimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_
↪ flexspi_nor_debug

```

For shield, please use the --shield to specify the shield to run, like

```

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

```

Sysbuild(System build) To support multicore project building, we ported Sysbuild from Zephyr. It supports combine multiple projects for compilation. You can build all projects by adding --sysbuild for main application. For example:

```

west build -b evkbnimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪ id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always

```

For more details, please refer to System build.

Config a Project Example in MCUXpresso SDK is configured and tested with pre-defined configuration. You can follow steps blow to change the configuration.

1. Run cmake configuration

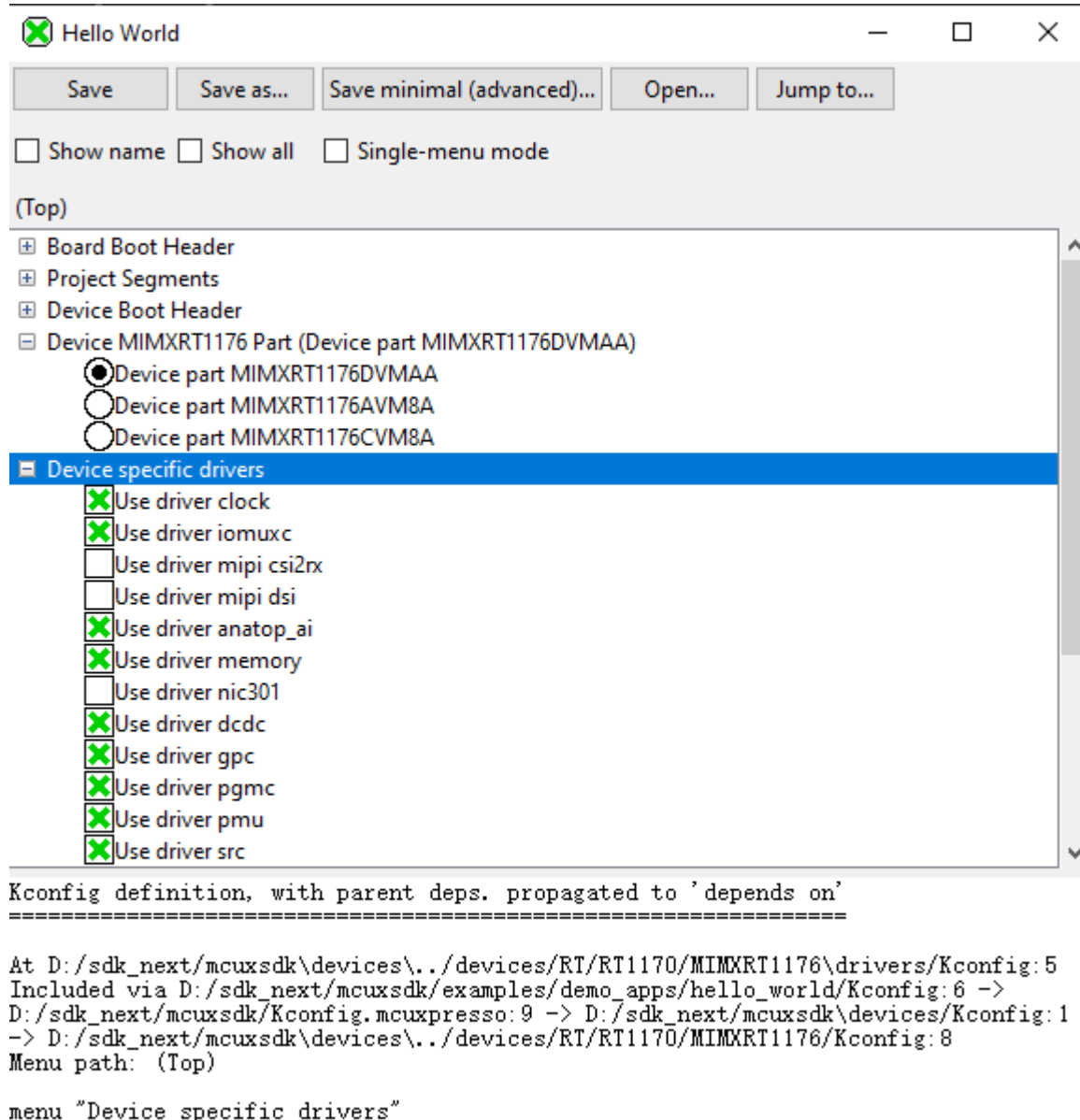
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

Please note the project will be built without --cmake-only parameter.

2. Run guiconfig target

```
west build -t guiconfig
```

Then you will get the Kconfig GUI launched, like



You can reconfigure the project by selecting/deselecting Kconfig options.

After saving and closing the Kconfig GUI, you can directly run `west build` to build with the new configuration.

Flash *Note:* Please refer Flash and Debug The Example to enable west flash/debug support. Flash the hello_world example:

```
west flash -r linkserver
```

Debug Start a gdb interface by following command:

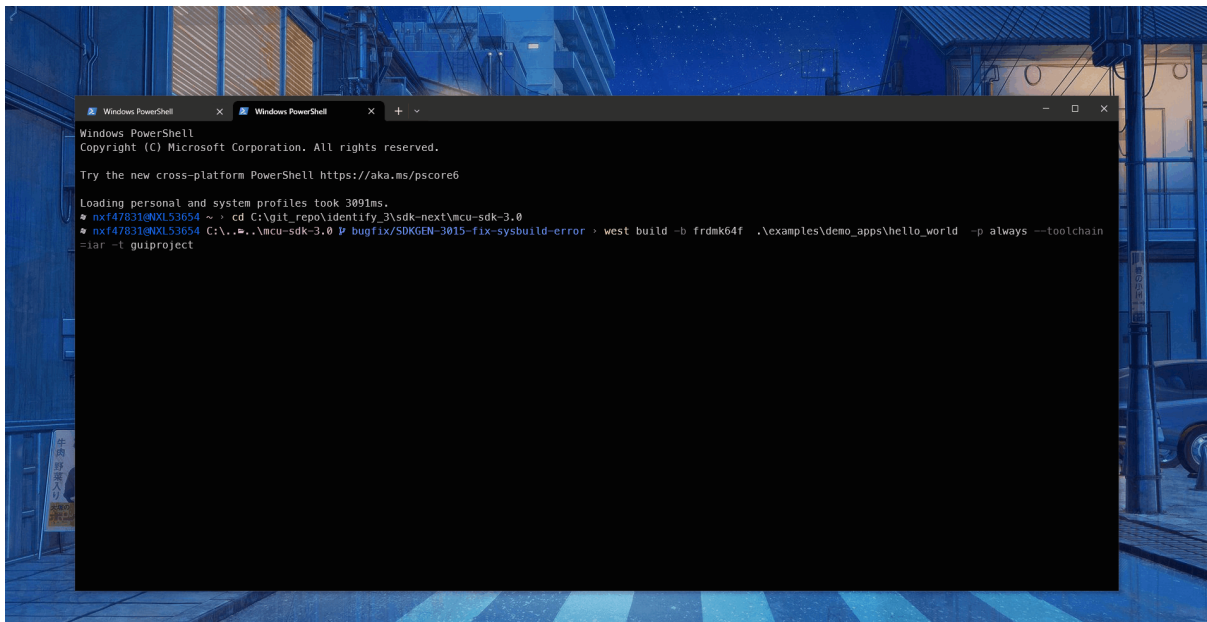
```
west debug -r linkserver
```

Work with IDE Project The above build functionalities are all with CLI. If you want to use the toolchain IDE to work to enjoy the better user experience especially for debugging or you are already used to develop with IDEs like IAR, MDK, Xtensa and CodeWarrior in the embedded world, you can play with our IDE project generation functionality.

This is the cmd to generate the evkbmimxrt1170 hello_world IAR IDE project files.

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_↵
↵flexspi_nor_debug -p always -t guiproject
```

By default, the IDE project files are generated in mcuxsdk/build/<toolchain> folder, you can open the project file with the IDE tool to work:

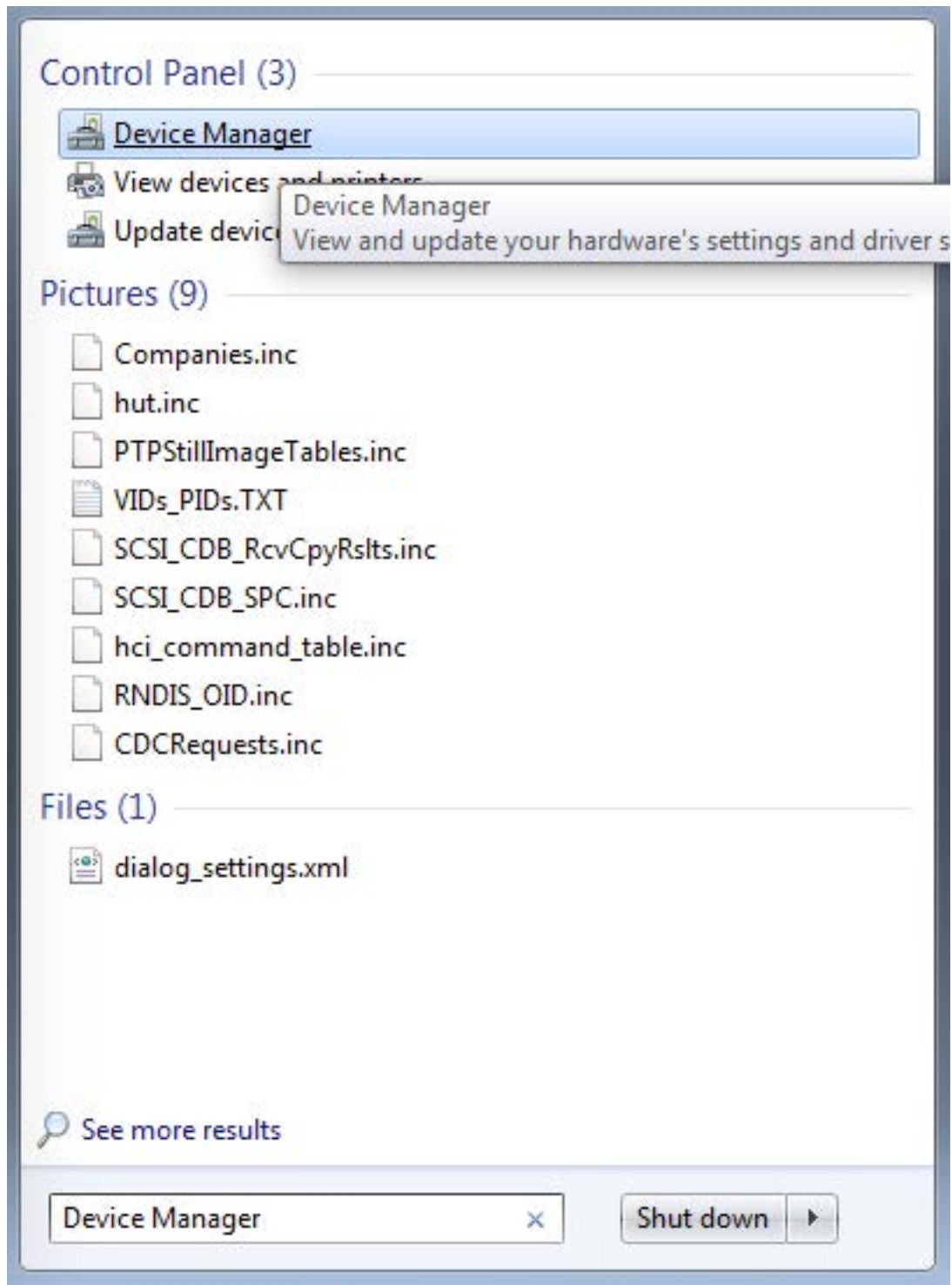


Note, please follow the [Installation](#) to setup the environment especially make sure that [ruby](#) has been installed.

1.2.2 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.

1.2.3 Updating debugger firmware

The MCXW23-EVK board comes with a CMSIS-DAP-compatible debug interface (known as MCU-Link). This firmware in this debug interface may be updated using the host computer scripts. This typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to re-program the debug probe firmware.

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

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

1. Install the MCU-Link utility.
2. Unplug the board's USB cable.
3. Install the jumper on J32.
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 J32.
7. Re-power the board by removing the USB cable and plugging it in again.

1.3 Release Notes

This is an Ready For Production Release (RFP) for MCXW23-EVK development board.

1.3.1 MCUXpresso SDK Release Notes

Overview

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

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

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

MCUXpresso SDK

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

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

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

Development tools

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

- MCUXpresso for VS Code v25.06
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

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

Devel- opment boards	MCU devices
MCXW23-EVK	MCXW235AIHNAR, MCXW235AIUKAR, MCXW235BIHNAR, MCXW235BIUKAR, MCXW236AIHNAR, MCXW236AIUKAR, MCXW236BIHNAR , MCXW236BIUKAR

MCUXpresso SDK release package

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

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

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

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

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

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

mbedTLS mbedtls SSL/TLS library v3.x

TF-M Trusted Firmware - M Library

PSA Test Suite Arm Platform Security Architecture Test Suite

Wireless Bluetooth LE host stack and applications The Bluetooth LE Host Stack component provides an implementation for a Bluetooth LE mandatory and some optional, proprietary, and experimental features. The Bluetooth LE Host Stack component provides application examples, services, and profiles.

Main features supported:

- Automotive Compliance
- MISRA Compliance
- HIS CCM <= 20
- Advanced Secure Mode
- Enhanced ATT
- GATT Caching
- Bluetooth LE Host GCC Libraries
- Bluetooth LE Host IAR Libraries
- Bluetooth LE Host Peripheral Libraries
- Bluetooth LE Central Libraries
- Bluetooth LE Host Full Host Features Libraries
- Bluetooth LE Host Optional Features Libraries
- Bluetooth LE Host Mandatory Features Libraries
- Bare-metal and FreeRTOS Support
- Bluetooth LE Privacy Support
- CCC Sample Applications
- Enhanced Notifications

- Dynamic Database
- OTA Support - Sample Applications
- Decision based Advertising Filtering (DBAF) - Experimental feature
- Advertising Coding Selection (ACS) - Experimental feature
- Channel Sounding - Experimental feature with controlled access (contact your NXP representative for access)
- Bluetooth LE Controller main and experimental features and capabilities described below are supported by the Bluetooth LE Host.

Note: For evaluating DBAF and ACS experimental features, replace the Bluetooth LE Host default example projects libraries with the libraries from the *SDK* folder *..\middleware\wireless\bluetooth\host\lib_exp* and enable the features in the application. The Radio Subsystem (NBU) Firmware with experimental features is required.

Wireless 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, such as, Low power, Over the Air (OTA) Firmware update, File System, Security, Sensors, Serial Connectivity Interface (FSCI), and others. The Connectivity Framework modules are located in the *middleware\wireless\framework* *SDK* folder.

Bluetooth Synopsys Controller

- Main features supported:
 - All roles that the Bluetooth specification specifies:
 - * Broadcaster
 - * Observer
 - * Peripheral
 - * Central
 - Up to 4 simultaneous connections supported
 - Bluetooth Low Energy features:
 - * Device privacy and network privacy modes (version 5.0)
 - * Advertising extension PDUs (version 5.0)
 - * Anonymous device address type (version 5.0)
 - * Up to 2 Mbps data rate (version 5.0)
 - * Long range (version 5.0)
 - * High-duty cycle, nonconnectable advertising (version 5.0)
 - * Channel selection algorithm #2 (version 5.0)
 - * High output power (version 5.0)
 - * Advertising channel index (version 5.1)
 - * Periodic advertising sync transfer (PAST) (version 5.1)
 - * Supports LE power control feature (version 5.2)
 - Device filtering through programmable size white lists
 - Direct test mode

- RF antenna: 50 Ω single-ended
- RF receiver characteristics:
 - * Sensitivity \approx 94 dBm in Bluetooth Low Energy 2 Mbps
 - * Sensitivity \approx 97 dBm in Bluetooth Low Energy 1 Mbps
 - * Sensitivity \approx 100 dBm in Bluetooth Low Energy 500 kbps
 - * Sensitivity \approx 102 dBm in Bluetooth Low Energy 125 kbps
 - * Accurate RSSI measurement with \pm 3 dB accuracy
- Flexible RF transmitter level configurability:
 - * TX mode 1 (TXM1): Range from \approx 31 dBm to +2 dBm when VDD_RF exceeds 1.1 V
 - * TX mode 2 (TXM2): Range from \approx 28 dBm to +6 dBm when VDD_RF exceeds 1.7 V

LittleFS LittleFS filesystem stack

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 (25.03.00).

- **Bluetooth Synopsys controller**

Added

- Initial version of Synopsys link layer added to enable Bluetooth use cases for the MCXW23 family.

- Support for **HCI black box** application added.
- **Health Care Iot Reference design applications**

Added

- Initial version of Low Power Health Care Iot Peripheral application added.
- Initial version of Health Care Iot Central application added.
- **Bluetooth LE host stack and applications**

Added

- Support for **Wireless UART** application added.
- Support for **Firmware Over The Air** applications added (using ATT and L2CAP).

Known issues

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

Limitations when creating a new FreeRTOS-based C/C++ project

Due to the missing component dependencies definition, there are several limitations when creating a new FreeRTOS-based C/C++ project in MCUXpresso IDE. When the **FreeRTOS kernel** component is selected (under Operating Systems/RTOS/Core menu), you must manually select the **FreeRTOS cm33 non trustzone port** component (under Middleware/RTOS menu) for projects without TrustZone. For FreeRTOS TrustZone projects creation, the support is not ready.

Wireless UART application – Bluetooth Low Energy advertising and connection loss issue

When using the Wireless UART application with default settings, functionality is as expected. However, the following issue occurs when the Bluetooth Low Energy advertising interval is set to 20 milliseconds and the connection interval is set to 7.5 milliseconds: After two devices establish a connection, the central device fails to start advertising to a third device after a button press. The HCI command to start advertising returns success, but the device does not transmit any advertising packets. Additionally, the supervision timeout causes the existing connection to drop unexpectedly.

Bluetooth Synopsys Controller

- Stability observation during extended testing The `llhwc_set_adv_param` function shows unexpected behavior during extended sequences of link layer tests, typically after 1.5 hours of continuous execution without a hardware reset.
 - This rare behavior occurs only under specific test conditions.
 - The behavior relates to the extended advertising feature.
 - This behavior does not impact regular usage scenarios.
- Faulty passive channel assessment behavior
 - Connection establishment fails when channel assessment finds only one suitable channel. However, the failure occurs rarely.

- Channel assessment fails on connections with slave latency greater than zero.

1.4 ChangeLog

1.4.1 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
-

ANACTRL

[2.4.0]

- Improvements
 - Added some interrupt flags for devices containing BOD1 and BOD2 interrupt controls.
 - Added a control macro to enable/disable the 32MHz Crystal oscillator code in current driver.
 - Added a feature macro for bit field ENA_96MHZCLK in FRO192M_CTRL.
 - Added a feature macro for bit field BODCORE_INT_ENABLE in BOD_DCDC_INT_CTRL.

[2.3.1]

- Bug Fixes
 - Added casts to prevent overflow caused by capturing large target clock.

[2.3.0]

- Improvements
 - Added AUX_BIAS control APIs.

[2.2.0]

- Improvements
 - Added some macros to separate the scenes that some bit fields are reserved for some devices.
 - Optimized the comments.
 - Optimized the code implementation inside some functions.

[2.1.2]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.3 and rule 17.7.

[2.1.1]

- Bug Fixes
 - Removed AnalogTestBus configuration to align with new header.

[2.1.0]

- Improvements
 - Updates for LPC55xx A1.
 - * Removed the control of bitfield FRO192M_CTRL_ENA_48MHZCLK, XO32M_CTRL_ACBUF_PASS_ENABLE.
 - * Removed status bits in ANACTRL_STATUS: PMU_ID OSC_ID FINAL_TEST_DONE_VECT.
 - * Removed API ANACTRL_EnableAdcVBATDivider() and APIs which operate the RingOSC registers.
 - * Removed the configurations of 32 MHz Crystal oscillator voltage source supply control register.
 - * Added API ANACTRL_ClearInterrupts().

[2.0.0]

- Initial version.
-

CASSPER**[2.2.4]**

- Fix MISRA-C 2012 issue.

[2.2.3]

- Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.

[2.2.2]

- Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE

[2.2.1]

- Fix MISRA C-2012 issue.

[2.2.0]

- Rework driver to support multiple curves at once.

[2.1.0]

- Add ECC NIST P-521 elliptic curve.

[2.0.10]

- Fix MISRA C-2012 issue.

[2.0.9]

- Remove unused function Jac_oncurve().
- Fix ECC384 build.

[2.0.8]

- Add feature macro for CASPER_RAM_OFFSET.

[2.0.7]

- Fix MISRA C-2012 issue.

[2.0.6]

- Bug Fixes
 - Fix IAR Pa082 warning

[2.0.5]

- Bug Fixes
 - Fix sign-compare warning

[2.0.4]

- For GCC compiler, enforce O1 optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires -fno-strict-aliasing.

[2.0.3]

- Bug Fixes
 - Fixed the bug for KPSDK-28107 RSUB, FILL and ZERO operations not implemented in enum_casper_operation.

[2.0.2]

- Bug Fixes
 - Fixed KPSDK-25015 CASPER_MEMCPY hard-fault on LPC55xx when both source and destination buffers are outside of CASPER_RAM.

[2.0.1]

- Bug Fixes
 - Fixed the bug that KPSDK-24531 double_scalar_multiplication() result may be all zeroes for some specific input.

[2.0.0]

- Initial version.
-

CDOG**[2.1.3]**

- Re-design multiple instance IRQs and Clocks
- Add fix for RESTART command errata

[2.1.2]

- Support multiple IRQs
- Fix default CONTROL values

[2.1.1]

- Remove bit CONTROL[CONTROL_CTRL].

[2.1.0]

- Rename CWT to CDOG.

[2.0.2]

- Fix MISRA-2012 issues.

[2.0.1]

- Fix doxygen issues.

[2.0.0]

- Initial version.
-

CLOCK

[2.1.2]

- Improvements
 - Include the fsl_common.h in clock driver to fix the compiling issue in Zephyr build.

[2.1.1]

- Improvements
 - Fixed the coverity high issues in clock driver. Before using the pointer, check the passed parameter to ensure that it will not corrupt or misinterpret adjacent memory locations.

[2.1.0]

- New Features
 - Added CLOCK_SetupPLUClkInClocking() to store the PLU CLKIN frequency.
- Bug Fixes
 - Fixed CLOCK_GetFlexCommClkFreq function to get flexcomm output frequency correctly.
 - Fixed attach incorrect attach_id for SCT.

[2.0.3]

- Bug Fixes
 - Fixed attach incorrect attach_id.

[2.0.2]

- New Features
 - Added get actual clock attach id api to allow users to obtain the actual clock source in target register.
- Bug Fixes
 - The attach clock and get actual clock attach id APIs should check combination of two clock sources.
- Optimizations
 - Made the judgement statements more clear.
 - Strengthened the compatibility of clock attach id.
 - Removed some unmeaningful definitions and add some useful ones to enhance readability.

[2.0.1]

- Some minor fixes.

[2.0.0]

- Initial version.
-

COMMON**[2.6.0]**

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.3]

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

[2.3.2]

- Improvements
 - Make driver `aarch64` compatible

[2.3.1]

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

[2.3.0]

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

[2.2.10]

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

[2.2.9]

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

[2.2.8]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

CRC

[2.1.1]

- Fix MISRA issue.

[2.1.0]

- Add CRC_WriteSeed function.

[2.0.2]

- Fix MISRA issue.

[2.0.1]

- Fixed KPSDK-13362. MDK compiler issue when writing to WR_DATA with -O3 optimize for time.

[2.0.0]

- Initial version.
-

CTIMER

[2.3.3]

- Bug Fixes
 - Fix CERT INT30-C INT31-C issue.
 - Make API CTIMER_SetupPwm and CTIMER_UpdatePwmDutycycle return fail if pulse width register overflow.

[2.3.2]

- Bug Fixes
 - Clear unexpected DMA request generated by RESET_PeripheralReset in API CTIMER_Init to avoid trigger DMA by mistake.

[2.3.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.7 and 12.2.

[2.3.0]

- Improvements
 - Added the CTIMER_SetPrescale(), CTIMER_GetCaptureValue(), CTIMER_EnableResetMatchChannel(), CTIMER_EnableStopMatchChannel(), CTIMER_EnableRisingEdgeCapture(), CTIMER_EnableFallingEdgeCapture(), CTIMER_SetShadowValue(), APIs Interface to reduce code complexity.

[2.2.2]

- Bug Fixes
 - Fixed SetupPwm() API only can use match 3 as period channel issue.

[2.2.1]

- Bug Fixes
 - Fixed use specified channel to setting the PWM period in SetupPwmPeriod() API.
 - Fixed Coverity Out-of-bounds issue.

[2.2.0]

- Improvements
 - Updated three API Interface to support Users to flexibly configure the PWM period and PWM output.
- Bug Fixes
 - MISRA C-2012 issue fixed: rule 8.4.

[2.1.0]

- Improvements
 - Added the CTIMER_GetOutputMatchStatus() API Interface.
 - Added feature macro for FSL_FEATURE_CTIMER_HAS_NO_CCR_CAP2 and FSL_FEATURE_CTIMER_HAS_NO_IR_CR2INT.

[2.0.3]

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

[2.0.2]

- New Features
 - Added new API “CTIMER_GetTimerCountValue” to get the current timer count value.
 - Added a control macro to enable/disable the RESET and CLOCK code in current driver.
 - Added a new feature macro to update the API of CTimer driver for lpc8n04.

[2.0.1]

- Improvements
 - API Interface Change
 - * Changed API interface by adding CTIMER_SetupPwmPeriod API and CTIMER_UpdatePwmPulsePeriod API, which both can set up the right PWM with high resolution.

[2.0.0]

- Initial version.
-

LPC_DMA

[2.5.3]

- Improvements
 - Add assert in DMA_SetChannelXferConfig to prevent XFERCOUNT value overflow.

[2.5.2]

- Bug Fixes
 - Use separate “SET” and “CLR” registers to modify shared registers for all channels, in case of thread-safe issue.

[2.5.1]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rule 11.6.

[2.5.0]

- Improvements
 - Added a new api DMA_SetChannelXferConfig to set DMA xfer config.

[2.4.4]

- Bug Fixes
 - Fixed the issue that DMA_IRQHandle might generate redundant callbacks.
 - Fixed the issue that DMA driver cannot support channel bigger then 32.
 - Fixed violation of the MISRA C-2012 rule 13.5.

[2.4.3]

- Improvements
 - Added features FSL_FEATURE_DMA_DESCRIPTOR_ALIGN_SIZEn/FSL_FEATURE_DMA0_DESCRIPTOR_ALIGN_SIZEn to support the descriptor align size not constant in the two instances.

[2.4.2]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rule 8.4.

[2.4.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 5.7, 8.3.

[2.4.0]

- Improvements
 - Added new APIs DMA_LoadChannelDescriptor/DMA_ChannelIsBusy to support polling transfer case.
- Bug Fixes
 - Added address alignment check for descriptor source and destination address.
 - Added DMA_ALLOCATE_DATA_TRANSFER_BUFFER for application buffer allocation.
 - Fixed the sign-compare warning.
 - Fixed violations of the MISRA C-2012 rules 18.1, 10.4, 11.6, 10.7, 14.4, 16.3, 20.7, 10.8, 16.1, 17.7, 10.3, 3.1, 18.1.

[2.3.0]

- Bug Fixes
 - Removed DMA_HandleIRQ prototype definition from header file.
 - Added DMA_IRQHandle prototype definition in header file.

[2.2.5]

- Improvements
 - Added new API DMA_SetupChannelDescriptor to support configuring wrap descriptor.
 - Added wrap support in function DMA_SubmitChannelTransfer.

[2.2.4]

- Bug Fixes
 - Fixed the issue that macro DMA_CHANNEL_CFER used wrong parameter to calculate DSTINC.

[2.2.3]

- Bug Fixes
 - Improved DMA driver Deinit function for correct logic order.
- Improvements
 - Added API DMA_SubmitChannelTransferParameter to support creating head descriptor directly.
 - Added API DMA_SubmitChannelDescriptor to support ping pong transfer.
 - Added macro DMA_ALLOCATE_HEAD_DESCRIPTOR/DMA_ALLOCATE_LINK_DESCRIPTOR to simplify DMA descriptor allocation.

[2.2.2]

- Bug Fixes
 - Do not use software trigger when hardware trigger is enabled.

[2.2.1]

- Bug Fixes
 - Fixed Coverity issue.

[2.2.0]

- Improvements
 - Changed API DMA_SetupDMADescriptor to non-static.
 - Marked APIs below as deprecated.
 - * DMA_PrepareTransfer.
 - * DMA_Submit transfer.
 - Added new APIs as below:
 - * DMA_SetChannelConfig.
 - * DMA_PrepareChannelTransfer.
 - * DMA_InstallDescriptorMemory.
 - * DMA_SubmitChannelTransfer.
 - * DMA_SetChannelConfigValid.

- * DMA_DoChannelSoftwareTrigger.
- * DMA_LoadChannelTransferConfig.

[2.0.1]

- Improvements
 - Added volatile for DMA descriptor member xfercfg to avoid optimization.

[2.0.0]

- Initial version.
-

FLEXCOMM

[2.0.2]

- Bug Fixes
 - Fixed typos in FLEXCOMM15_DriverIRQHandler().
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.
- Improvements
 - Added instance calculation in FLEXCOMM16_DriverIRQHandler() to align with Flexcomm 14 and 15.

[2.0.1]

- Improvements
 - Added more IRQHandler code in drivers to adapt new devices.

[2.0.0]

- Initial version.
-

GINT

[2.1.1]

- Improvements
 - Added support for platforms with PORT_POL and PORT_ENA registers without arrays.

[2.1.0]

- Improvements
 - Updated for platforms which only has one port.

[2.0.3]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.8.

[2.0.2]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 17.7.

[2.0.1]

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

[2.0.0]

- Initial version.
-

GPIO

[2.1.7]

- Improvements
 - Enhanced GPIO_PinInit to enable clock internally.

[2.1.6]

- Bug Fixes
 - Clear bit before set it within GPIO_SetPinInterruptConfig() API.

[2.1.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 3.1, 10.6, 10.7, 17.7.

[2.1.4]

- Improvements
 - Added API GPIO_PortGetInterruptStatus to retrieve interrupt status for whole port.
 - Corrected typos in header file.

[2.1.3]

- Improvements
 - Updated “GPIO_PinInit” API. If it has DIRCLR and DIRSET registers, use them at set 1 or clean 0.

[2.1.2]

- Improvements
 - Removed deprecated APIs.

[2.1.1]

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

[2.1.0]

- New Features
 - Added GPIO initialize API.

[2.0.0]

- Initial version.
-

HASHCRYPT**[2.0.0]**

- Initial version.

[2.0.1]

- Supported loading AES key from unaligned address.

[2.0.2]

- Supported loading AES key from unaligned address for different compiler and core variants.

[2.0.3]

- Remove SHA512 and AES ICB algorithm definitions

[2.0.4]

- Add SHA context switch support

[2.1.0]

- Update the register name and macro to align with new header.
- Fixed the sign-compare warning in `hashcrypt_load_data`.

[2.1.1]

- Fix MISRA C-2012.

[2.1.2]

- Support loading AES input data from unaligned address.

[2.1.3]

- Fix MISRA C-2012.

[2.1.4]

- Fix context switch cannot work when switching from AES.

[2.1.5]

- Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` to prevent possible optimization issue.

[2.2.0]

- Add AES-OFB and AES-CFB mixed IP/SW modes.

[2.2.1]

- Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` prevent compiler from reordering memory write when -O2 or higher is used.

[2.2.2]

- Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` to fix optimization issue

[2.2.3]

- Added check for size in `hashcrypt_aes_one_block` to prevent overflowing COUNT field in MEMCTRL register, if its bigger than COUNT field do a multiple runs.

[2.2.4]

- In all `HASHCRYPT_AES_xx` functions have been added setting CTRL_MODE bitfield to 0 after processing data, which decreases power consumption.

[2.2.5]

- Add data synchronization barrier and instruction synchronization barrier inside `hashcrypt_sha_process_message_data()` to fix optimization issue

[2.2.6]

- Add data synchronization barrier inside HASHCRYPT_SHA_Update() and hashcrypt_get_data() function to fix optimization issue on MDK and ARMGCC release targets

[2.2.7]

- Add data synchronization barrier inside HASHCRYPT_SHA_Update() to fix optimization issue on MCUX IDE release target

[2.2.8]

- Unify hashcrypt hashing behavior between aligned and unaligned input data

[2.2.9]

- Add handling of set ERROR bit in the STATUS register

[2.2.10]

- Fix missing error statement in hashcrypt_save_running_hash()

[2.2.11]

- Fix incorrect SHA-256 calculation for long messages with reload

[2.2.12]

- Fix hardfault issue on the Keil compiler due to unaligned memcpy() input on some optimization levels

[2.2.13]

- Added function hashcrypt_seed_prng() which loading random number into PRNG_SEED register before AES operation for SCA protection

[2.2.14]

- Modify function hashcrypt_get_data() to prevent issue with unaligned access

[2.2.15]

- Add wait on DIGEST BIT inside hashcrypt_sha_one_block() to fix issues with some optimization flags

[2.2.16]

- Add DSB instruction inside hashcrypt_sha_ldm_stm_16_words() to fix issues with some optimization flags

I2C

[2.3.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1.
 - Fixed issue that if master only sends address without data during I2C interrupt transfer, address nack cannot be detected.

[2.3.2]

- Improvement
 - Enable or disable timeout option according to enableTimeout.
- Bug Fixes
 - Fixed timeout value calculation error.
 - Fixed bug that the interrupt transfer cannot recover from the timeout error.

[2.3.1]

- Improvement
 - Before master transfer with transactional APIs, enable master function while disable slave function and vice versa for slave transfer to avoid the one affecting the other.
- Bug Fixes
 - Fixed bug in I2C_SlaveEnable that the slave enable/disable should not affect the other register bits.

[2.3.0]

- Improvement
 - Added new return codes kStatus_I2C_EventTimeout and kStatus_I2C_SclLowTimeout, and added the check for event timeout and SCL timeout in I2C master transfer.
 - Fixed bug in slave transfer that the address match event should be invoked before not after slave transmit/receive event.

[2.2.0]

- New Features
 - Added enumeration `_i2c_status_flags` to include all previous master and slave status flags, and added missing status flags.
 - Modified `I2C_GetStatusFlags` to get all I2C flags.
 - Added API `I2C_ClearStatusFlags` to clear all clearable flags not just master flags.
 - Modified master transactional APIs to enable bus event timeout interrupt during transfer, to avoid glitch on bus causing transfer hangs indefinitely.
- Bug Fixes
 - Fixed bug that status flags and interrupt enable masks share the same enumerations by adding enumeration `_i2c_interrupt_enable` for all master and slave interrupt sources.

[2.1.0]

- Bug Fixes
 - Fixed bug that during master transfer, when master is nacked during slave probing or sending subaddress, the return status should be `kStatus_I2C_Addr_Nak` rather than `kStatus_I2C_Nak`.
- Bug Fixes
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.4, 13.5.
- New Features
 - Added macro `I2C_MASTER_TRANSMIT_IGNORE_LAST_NACK`, so that user can configure whether to ignore the last byte being nacked by slave during master transfer.

[2.0.8]

- Bug Fixes
 - Fixed `I2C_MasterSetBaudRate` issue that `MSTSCLOW` and `MSTSCHIGH` are incorrect when `MSTTIME` is odd.

[2.0.7]

- Bug Fixes
 - Two dividers, `CLKDIV` and `MSTTIME` are used to configure baudrate. According to reference manual, in order to generate 400kHz baudrate, the clock frequency after `CLKDIV` must be less than 2mHz. Fixed the bug that, the clock frequency after `CLKDIV` may be larger than 2mHz using the previous calculation method.
 - Fixed MISRA 10.1 issues.
 - Fixed wrong baudrate calculation when feature `FSL_FEATURE_I2C_PREPCLKFRG_8MHZ` is enabled.

[2.0.6]

- New Features
 - Added master timeout self-recovery support for feature `FSL_FEATURE_I2C_TIMEOUT_RECOVERY`.
- Bug Fixes
 - Eliminated IAR Pa082 warning.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.

[2.0.5]

- Bug Fixes
 - Fixed wrong assignment for `datasize` in `I2C_InitTransferStateMachineDMA`.
 - Fixed wrong working flow in `I2C_RunTransferStateMachineDMA` to ensure master can work in no start flag and no stop flag mode.

- Fixed wrong working flow in I2C_RunTransferStateMachine and added kReceive-DataBeginState in _i2c_transfer_states to ensure master can work in no start flag and no stop flag mode.
- Fixed wrong handle state in I2C_MasterTransferDMAHandleIRQ. After all the data has been transfered or nak is returned, handle state should be changed to idle.
- Improvements
 - Rounded up the calculated divider value in I2C_MasterSetBaudRate.

[2.0.4]

- Improvements
 - Updated the I2C_WATI_TIMEOUT macro to unified name I2C_RETRY_TIMES
 - Updated the “I2C_MasterSetBaudRate” API to support baudrate configuration for feature QN9090.
- Bug Fixes
 - Fixed build warnning caused by uninitialized variable.
 - Fixed COVERITY issue of unchecked return value in I2C_RTOS_Transfer.

[2.0.3]

- Improvements
 - Unified the component full name to FLEXCOMM I2C(DMA/FREERTOS) driver.

[2.0.2]

- Improvements
 - In slave IRQ:
 1. Changed slave receive process to first set the I2C_SLVCTL_SLVCONTINUE_MASK to acknowledge the received data, then do data receive.
 2. Improved slave transmit process to set the I2C_SLVCTL_SLVCONTINUE_MASK immediately after writing the data.

[2.0.1]

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

[2.0.0]

- Initial version.
-

INPUTMUX

[2.0.9]

- Improvements
 - Use INPUTMUX_CLOCKS to initialize the inputmux module clock to adapt to multiple inputmux instances.
 - Modify the API base type from INPUTMUX_Type to void.

[2.0.8]

- Improvements
 - Updated a feature macro usage for function INPUTMUX_EnableSignal.

[2.0.7]

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

[2.0.6]

- Bug Fixes
 - Fixed the documentation wrong in API INPUTMUX_AttachSignal.

[2.0.5]

- Bug Fixes
 - Fixed build error because some devices has no sct.

[2.0.4]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rule 10.4, 12.2 in INPUTMUX_EnableSignal() function.

[2.0.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.4, 10.7, 12.2.

[2.0.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.4, 12.2.

[2.0.1]

- Support channel mux setting in INPUTMUX_EnableSignal().

[2.0.0]

- Initial version.
-

IOCON

[2.2.0]

- Improvements
 - Removed duplicate macro definitions.
 - Renamed 'IOCON_I2C_SLEW' macro to 'IOCON_I2C_MODE' to match its companion 'IOCON_GPIO_MODE'. The original is kept as a deprecated symbol.

[2.1.2]

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

[2.1.1]

- Updated left shift format with mask value instead of a constant value to automatically adapt to all platforms.

[2.1.0]

- Added a new IOCON_PinMuxSet() function with a feature IOCON_ONE_DIMENSION for LPC845MAX board.

[2.0.0]

- Initial version.
-

MAILBOX

[2.3.3]

- Improvements
 - Added support for the MCXN556S, MCXN537, MCXN536, MCXN527 and MCXN526 series

[2.3.2]

- Improvements
 - Added support for the MCXN946 and MCXN546 series

[2.3.1]

- Improvements
 - Added support for the LPC55S66 series.

[2.3.0]

- Improvements
 - Added support for the MCXNx4x series with new value for kMAILBOX_CM33_Core0 or kMAILBOX_CM33_Core1.

[2.2.0]

- Improvements
 - Fixed missing conditional defines for the LPC5411x series.

[2.1.0]

- Improvements
 - Added support for the LPC55S69 series. cpu_id parameter can be newly assigned to kMAILBOX_CM33_Core0 or kMAILBOX_CM33_Core1.

[2.0.0]

- Initial version.
-

MRT**[2.0.5]**

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.4]

- Improvements
 - Don't reset MRT when there is not system level MRT reset functions.

[2.0.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1 and 10.4.
 - Fixed the wrong count value assertion in MRT_StartTimer API.

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

OSTIMER

[2.2.5]

- Improvements
 - Support binary encoded ostimer.

[2.2.4]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.2.3]

- Improvements
 - Disable and clear pending interrupts before disabling the OSTIMER clock to avoid interrupts being executed when the clock is already disabled.

[2.2.2]

- Improvements
 - Support devices with different OSTIMER instance name.

[2.2.1]

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

[2.2.0]

- Improvements
 - Move the PMC operation out of the OSTIMER driver to board specific files.
 - Added low level APIs to control OSTIMER MATCH and interrupt.

[2.1.2]

- Bug Fixes
 - Fixed MISRA-2012 rule 10.8.

[2.1.1]

- Bug Fixes
 - removes the suffix 'n' for some register names and bit fields' names
- Improvements
 - Added HW CODE GRAY feature supported by CODE GRAY in SYSCTRL register group.

[2.1.0]

- Bug Fixes
 - Added a workaround to fix the issue that no interrupt was reported when user set smaller period.
 - Fixed violation of MISRA C-2012 rule 10.3 and 11.9.
- Improvements
 - Added return value for the two APIs to set match value.
 - * OSTIMER_SetMatchRawValue
 - * OSTIMER_SetMatchValue

[2.0.3]

- Bug Fixes
 - Fixed violation of MISRA C-2012 rule 10.3, 14.4, 17.7.

[2.0.2]

- Improvements
 - Added support for OSTIMER0

[2.0.1]

- Improvements
 - Removed the software reset function out of the initialization API.
 - Enabled interrupt directly instead of enabling deep sleep interrupt. Users need to enable the deep sleep interrupt in application code if needed.

[2.0.0]

- Initial version.
-

PINT**[2.2.0]**

- Fixed
 - Fixed the issue that clear interrupt flag when it's not handled. This causes events to be lost.
- Changed
 - Used one callback for one PINT instance. It's unnecessary to provide different callbacks for all PINT events.

[2.1.13]

- Improvements
 - Added instance array for PINT to adapt more devices.
 - Used release reset instead of reset PINT which may clear other related registers out of PINT.

[2.1.12]

- Bug Fixes
 - Fixed coverity issue.

[2.1.11]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.7 violation.

[2.1.10]

- New Features
 - Added the driver support for MCXN10 platform with combined interrupt handler.

[2.1.9]

- Bug Fixes
 - Fixed MISRA-2012 rule 8.4.

[2.1.8]

- Bug Fixes
 - Fixed MISRA-2012 rule 10.1 rule 10.4 rule 10.8 rule 18.1 rule 20.9.

[2.1.7]

- Improvements
 - Added fully support for the SECPINT, making it can be used just like PINT.

[2.1.6]

- Bug Fixes
 - Fixed the bug of not enabling common pint clock when enabling security pint clock.

[2.1.5]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 10.1 rule 10.3 rule 10.4 rule 10.8 rule 14.4.
 - Changed interrupt init order to make pin interrupt configuration more reasonable.

[2.1.4]

- Improvements
 - Added feature to control distinguish PINT/SECPINT relevant interrupt/clock configurations for PINT_Init and PINT_Deinit API.
 - Swapped the order of clearing PIN interrupt status flag and clearing pending NVIC interrupt in PINT_EnableCallback and PINT_EnableCallbackByIndex function.
 - Bug Fixes
 - * Fixed build issue caused by incorrect macro definitions.

[2.1.3]

- Bug fix:
 - Updated PINT_PinInterruptClrStatus to clear PINT interrupt status when the bit is asserted and check whether was triggered by edge-sensitive mode.
 - Write 1 to IST corresponding bit will clear interrupt status only in edge-sensitive mode and will switch the active level for this pin in level-sensitive mode.
 - Fixed MISRA c-2012 rule 10.1, rule 10.6, rule 10.7.
 - Added FSL_FEATURE_SECPINT_NUMBER_OF_CONNECTED_OUTPUTS to distinguish IRQ relevant array definitions for SECPINT/PINT on lpc55s69 board.
 - Fixed PINT driver c++ build error and remove index offset operation.

[2.1.2]

- Improvement:
 - Improved way of initialization for SECPINT/PINT in PINT_Init API.

[2.1.1]

- Improvement:
 - Enabled secure pint interrupt and add secure interrupt handle.

[2.1.0]

- Added PINT_EnableCallbackByIndex/PINT_DisableCallbackByIndex APIs to enable/disable callback by index.

[2.0.2]

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

[2.0.1]

- Bug fix:
 - Updated PINT driver to clear interrupt only in Edge sensitive.

[2.0.0]

- Initial version.
-

PLU

[2.2.1]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.3 and rule 17.7.

[2.2.0]

- Bug Fixes
 - Fixed wrong parameter of the PLU_EnableWakeIntRequest function.

[2.1.0]

- New Features
 - Added 4 new APIs to support Niobe4's wake-up/interrupt control feature, including PLU_GetDefaultWakeIntConfig(), PLU_EnableWakeIntRequest(), PLU_LatchInterrupt() and PLU_ClearLatchedInterrupt().
- Other Changes
 - Changed the register name LUT_INP to LUT_INP_MUX due to register map update.

[2.0.1]

- New Features
 - Added control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.0]

- Initial version.
-

PUF

[2.2.0]

- Add support for kPUF_KeySlot4.
- Add new PUF_ClearKey() function, that clears a desired PUF internal HW key register.

[2.1.6]

- Changed wait time in PUF_Init(), when initialization fails it will try PUF_Powercycle() with shorter time. If this shorter time will also fail, initialization will be tried with worst case time as before.

[2.1.5]

- Use common SDK delay in `puf_wait_usec()`.

[2.1.4]

- Replace register `uint32_t ticksCount` with `volatile uint32_t ticksCount` in `puf_wait_usec()` to prevent optimization out delay loop.

[2.1.3]

- Fix MISRA C-2012 issue.

[2.1.2]

- Update: Add automatic big to little endian swap for user (pre-shared) keys destined to secret hardware bus (PUF key index 0).

[2.1.1]

- Fix ARMGCC build warning .

[2.1.0]

- Align driver with PUF SRAM controller registers on LPCXpresso55s16.
- Update initialization logic .

[2.0.3]

- Fix MISRA C-2012 issue.

[2.0.2]

- New feature:
 - Add PUF configuration structure and support for PUF SRAM controller.
- Improvements:
 - Remove magic constants.

[2.0.1]

- Bug Fixes:
 - Fixed `puf_wait_usec` function optimization issue.

[2.0.0]

- Initial version.
-

RESET

[2.0.1]

- Improvements
 - Add RESET_ReleasePeripheralReset API.

[2.0.0]

- Initial version.
-

RTC

[2.2.0]

- New Features
 - Created new APIs for the RTC driver.
 - * RTC_EnableSubsecCounter
 - * RTC_GetSubsecValue

[2.1.3]

- Bug Fixes
 - Fixed issue that RTC_GetWakeupCount may return wrong value.

[2.1.2]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.1, 10.4 and 10.7.

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3 and 11.9.

[2.1.0]

- Bug Fixes
 - Created new APIs for the RTC driver.
 - * RTC_EnableTimer
 - * RTC_EnableWakeUpTimerInterruptFromDPD
 - * RTC_EnableAlarmTimerInterruptFromDPD
 - * RTC_EnableWakeupTimer
 - * RTC_GetEnabledWakeupTimer
 - * RTC_SetSecondsTimerMatch
 - * RTC_GetSecondsTimerMatch

- * RTC_SetSecondsTimerCount
- * RTC_GetSecondsTimerCount
- deprecated legacy APIs for the RTC driver.
 - * RTC_StartTimer
 - * RTC_StopTimer
 - * RTC_EnableInterrupts
 - * RTC_DisableInterrupts
 - * RTC_GetEnabledInterrupts

[2.0.0]

- Initial version.
-

SCTIMER

[2.5.1]

- Bug Fixes
 - Fixed bug in SCTIMER_SetupCaptureAction: When kSCTIMER_Counter_H is selected, events 12-15 and capture registers 12-15 CAPn_H field can't be used.

[2.5.0]

- Improvements
 - Add SCTIMER_GetCaptureValue API to get capture value in capture registers.

[2.4.9]

- Improvements
 - Supported platforms which don't have system level SCTIMER reset.

[2.4.8]

- Bug Fixes
 - Fixed the issue that the SCTIMER_UpdatePwmDutycycle() can't writes MATCH_H bit and RELOADn_H.

[2.4.7]

- Bug Fixes
 - Fixed the issue that the SCTIMER_UpdatePwmDutycycle() can't configure 100% duty cycle PWM.

[2.4.6]

- Bug Fixes
 - Fixed the issue where the H register was not written as a word along with the L register.
 - Fixed the issue that the SCTIMER_SetCOUNTValue() is not configured with high 16 bits in unify mode.

[2.4.5]

- Bug Fixes
 - Fix SCT_EV_STATE_STATEMSKn macro build error.

[2.4.4]

- Bug Fixes
 - Fix MISRA C-2012 issue 10.8.

[2.4.3]

- Bug Fixes
 - Fixed the wrong way of writing CAPCTRL and REGMODE registers in SCTIMER_SetupCaptureAction.

[2.4.2]

- Bug Fixes
 - Fixed SCTIMER_SetupPwm 100% duty cycle issue.

[2.4.1]

- Bug Fixes
 - Fixed the issue that MATCHn_H bit and RELOADn_H bit could not be written.

[2.4.0]

[2.3.0]

- Bug Fixes
 - Fixed the potential overflow issue of pulseperiod variable in SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle API.
 - Fixed the issue of SCTIMER_CreateAndScheduleEvent API does not correctly work with 32 bit unified counter.
 - Fixed the issue of position of clear counter operation in SCTIMER_Init API.
- Improvements
 - Update SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle to support generate 0% and 100% PWM signal.
 - Add SCTIMER_SetupEventActiveDirection API to configure event activity direction.

- Update SCTIMER_StartTimer/SCTIMER_StopTimer API to support start/stop low counter and high counter at the same time.
- Add SCTIMER_SetCounterState/SCTIMER_GetCounterState API to write/read counter current state value.
- Update APIs to make it meaningful.
 - * SCTIMER_SetEventInState
 - * SCTIMER_ClearEventInState
 - * SCTIMER_GetEventInState

[2.2.0]

- Improvements
 - Updated for 16-bit register access.

[2.1.3]

- Bug Fixes
 - Fixed the issue of uninitialized variables in SCTIMER_SetupPwm.
 - Fixed the issue that the Low 16-bit and high 16-bit work independently in SCTIMER driver.
- Improvements
 - Added an enumerable macro of unify counter for user.
 - * kSCTIMER_Counter_U
 - Created new APIs for the RTC driver.
 - * SCTIMER_SetupStateLdMethodAction
 - * SCTIMER_SetupNextStateActionwithLdMethod
 - * SCTIMER_SetCOUNTValue
 - * SCTIMER_GetCOUNTValue
 - * SCTIMER_SetEventInState
 - * SCTIMER_ClearEventInState
 - * SCTIMER_GetEventInState
 - Deprecated legacy APIs for the RTC driver.
 - * SCTIMER_SetupNextStateAction

[2.1.2]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7, 11.9, 14.2 and 15.5.

[2.1.1]

- Improvements
 - Updated the register and macro names to align with the header of devices.

[2.1.0]

- Bug Fixes
 - Fixed issue where SCT application level Interrupt handler function is occupied by SCT driver.
 - Fixed issue where wrong value for INSYNC field inside SCTIMER_Init function.
 - Fixed issue to change Default value for INSYNC field inside SCTIMER_GetDefaultConfig.

[2.0.1]

- New Features
 - Added control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.0]

- Initial version.
-

SPI

[2.3.2]

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

[2.3.1]

- Improvements
 - Changed SPI_DUMMYDATA to 0x00.

[2.3.0]

- Update version.

[2.2.2]

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

[2.2.1]

- Bug Fixes
 - Fixed MISRA 2012 10.4 issue.
 - Added code to clear FIFOs before transfer using DMA.

[2.2.0]

- Bug Fixes
 - Fixed bug that slave gets stuck during interrupt transfer.

[2.1.1]

- Improvements
 - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
 - Fixed MISRA 10.1, 5.7 issues.

[2.1.0]

- Bug Fixes
 - Fixed Coverity issue of incrementing null pointer in SPI_TransferHandleIRQInternal.
 - Eliminated IAR Pa082 warnings.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.
- New Features
 - Modified the definition of SPI_SSELPOL_MASK to support the socs that have only 3 SSEL pins.

[2.0.4]

- Bug Fixes
 - Fixed the bug of using read only mode in DMA transfer. In DMA transfer mode, if transfer->txData is NULL, code attempts to read data from the address of 0x0 for configuring the last frame.
 - Fixed wrong assignment of handle->state. During transfer handle->state should be kSPI_Busy rather than kStatus_SPI_Busy.
- Improvements
 - Rounded up the calculated divider value in SPI_MasterSetBaud.

[2.0.3]

- Improvements
 - Added “SPI_FIFO_DEPTH(base)” with more definition.

[2.0.2]

- Improvements
 - Unified the component full name to FLEXCOMM SPI(DMA/FREERTOS) driver.

[2.0.1]

- Changed the data buffer from uint32_t to uint8_t which matches the real applications for SPI DMA driver.
- Added dummy data setup API to allow users to configure the dummy data to be transferred.
- Added new APIs for half-duplex transfer function. Users can not only send and receive data by one API in polling/interrupt/DMA way, but choose either to transmit first or to receive first. Besides, the PCS pin can be configured as assert status in transmission (between transmit and receive) by setting the isPcsAssertInTransfer to true.

[2.0.0]

- Initial version.
-

SPI_DMA

[2.2.1]

- Bug Fixes
 - Fixed MISRA 2012 11.6 issue..

[2.2.0]

- Improvements
 - Supported dataSize larger than 1024 data transmit.
-

SPI Flash Interface

[2.0.3]

- Bug Fixes
- MISRA C-2012 issue fixed: rule 10.3, 10.4, and 14.4.

[2.0.2]

- Bug Fixes
 - Fixed the command function set issue. After the command being set, there will be no wait for the CMD flag, as it may have been cleared by CS deassert.

[2.0.1]

- New Features
 - Added an API to read/write 1/2 Bytes data from/to SPIFI. This interface is useful for flash command, which only needs 1/2 Bytes data. The previous driver needed users to make sure of the minimum length being 4, which might cause issues in some flash commands.

[2.0.0]

- Initial version.
-

TRNG

[2.0.18]

- Bug fix:
 - TRNG health checks now done in software on RT5xx and RT6xx.
-

[2.0.17]

- New features:
 - Add support for RT700.

[2.0.16]

- Improvements:
 - Added support for Dual oscillator mode.

[2.0.15]

- Other changes:
 - Changed TRNG_USER_CONFIG_DEFAULT_XXX values according to latest recommended by design team.

[2.0.14]

- New features:
 - Add support for RW610 and RW612.

[2.0.13]

- Bug fix:
 - After deepsleep it might return error, added clearing bits in TRNG_GetRandomData() and generating new entropy.
 - Modified reloading entropy in TRNG_GetRandomData(), for some data length it doesn't reloading entropy correctly.

[2.0.12]

- Bug fix:
 - For KW34A4_SERIES, KW35A4_SERIES, KW36A4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.

[2.0.11]

- Bug fix:
 - Add clearing pending errors in TRNG_Init().

[2.0.10]

- Bug Fix:
 - Fixed doxygen issues.

[2.0.9]

- Bug Fix:
 - Fix HIS_CCM metrics issues.

[2.0.8]

- Bug fix:
 - For K32L2A41A_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv4.

[2.0.7]

- Bug fix:
 - Fix MISRA 2004 issue rule 12.5.

[2.0.6]

- Bug fix:
 - For KW35Z4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.

[2.0.5]

- Improvements:
 - For FRQMIN, FRQMAX and OSCDIV, add possibility to use device specific preprocessor macro to define default value in TRNG user configuration structure.

[2.0.4]

- Bug Fix:
 - Fix MISRA-2012 issues.
 - * Rule 10.1, rule 10.3, rule 13.5, rule 16.1.

[2.0.3]

- Improvements:
 - update TRNG_Init to restart new entropy generation.

[2.0.2]

- Improvements:
 - fix MISRA issues
 - * Rule 14.4.

[2.0.1]

- New features:
 - Set default OSCDIV for Kinetis devices KL8x and KL28Z.
- Other changes:
 - Changed default OSCDIV for K81 to divide by 2.

[2.0.0]

- Initial version.
-

USART**[2.8.5]**

- Bug Fixes
 - Fixed race condition during call of USART_EnableTxDMA and USART_EnableRxDMA.

[2.8.4]

- Bug Fixes
 - Fixed exclusive access in USART_TransferReceiveNonBlocking and USART_TransferSendNonBlocking.

[2.8.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3, 11.8.

[2.8.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 14.2.

[2.8.1]

- Bug Fixes
 - Fixed the Baud Rate Generator(BRG) configuration in 32kHz mode.

[2.8.0]

- New Features
 - Added the rx timeout interrupts and status flags of bus status.
 - Added new rx timeout configuration item in usart_config_t.
 - Added API USART_SetRxTimeoutConfig for rx timeout configuration.
- Improvements
 - When the calculated baudrate cannot meet user's configuration, lower OSR value is allowed to use.

[2.7.0]

- New Features
 - Added the missing interrupts and status flags of bus status.
 - Added the check of tx error, noise error framing error and parity error in interrupt handler.

[2.6.0]

- Improvements
 - Used separate data for TX and RX in `usart_transfer_t`.
- Bug Fixes
 - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling `USART_TransferReceiveNonBlocking`, the received data count returned by `USART_TransferGetReceiveCount` is wrong.
- New Features
 - Added missing API `USART_TransferGetSendCountDMA` get send count using DMA.

[2.5.0]

- New Features
 - Added APIs `USART_GetRxFifoCount`/`USART_GetTxFifoCount` to get rx/tx FIFO data count.
 - Added APIs `USART_SetRxFifoWatermark`/`USART_SetTxFifoWatermark` to set rx/tx FIFO water mark.
- Bug Fixes
 - Fixed DMA transfer blocking issue by enabling tx idle interrupt after DMA transmission finishes.

[2.4.0]

- New Features
 - Modified `usart_config_t`, `USART_Init` and `USART_GetDefaultConfig` APIs so that the hardware flow control can be enabled during module initialization.
- Bug Fixes
 - Fixed MISRA 10.4 violation.

[2.3.1]

- Bug Fixes
 - Fixed bug that operation on `INTENSET`, `INTENCLR`, `FIFOINTENSET` and `FIFOINTENCLR` should use bitwise operation not ‘or’ operation.
 - Fixed bug that if rx interrupt occurs before TX interrupt is enabled and after `txDataSize` is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.
- Improvements
 - Added check for baud rate’s accuracy that returns `kStatus_USART_BaudrateNotSupport` when the best achieved baud rate is not within 3% error of configured baud rate.

[2.3.0]

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

- Modified USART_TransferReceiveNonBlocking and USART_TransferHandleIRQ to use 9-bit mode in multi-slave system.

[2.2.0]

- New Features
 - Added the feature of supporting USART working at 32 kHz clocking mode.
- Improvements
 - Modified USART_TransferHandleIRQ so that txState will be set to idle only when all data has been sent out to bus.
 - Modified USART_TransferGetSendCount so that this API returns the real byte count that USART has sent out rather than the software buffer status.
 - Added timeout mechanism when waiting for certain states in transfer driver.
- Bug Fixes
 - Fixed MISRA 10.1 issues.
 - Fixed bug that operation on INTENSET, INTENCLR, FIFOINTENSET and FIFOINTENCLR should use bitwise operation not 'or' operation.
 - Fixed bug that if rx interrupt occurs before TX interrupt is enabled and after txData-Size is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.

[2.1.1]

- Improvements
 - Added check for transmitter idle in USART_TransferHandleIRQ and USART_TransferSendDMACallback to ensure all the data would be sent out to bus.
 - Modified USART_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.
- Bug Fixes
 - Eliminated IAR Pa082 warnings.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.

[2.1.0]

- New Features
 - Added features to allow users to configure the USART to synchronous transfer(master and slave) mode.
- Bug Fixes
 - Modified USART_SetBaudRate to get more accurate configuration.

[2.0.3]

- New Features
 - Added new APIs to allow users to enable the CTS which determines whether CTS is used for flow control.

[2.0.2]

- Bug Fixes
 - Fixed the bug where transfer abort APIs could not disable the interrupts. The FIFOINTENSET register should not be used to disable the interrupts, so use the FIFOINTENCLR register instead.

[2.0.1]

- Improvements
 - Unified the component full name to FLEXCOMM USART (DMA/FREERTOS) driver.

[2.0.0]

- Initial version.
-

USART_DMA

[2.6.0]

- Refer USART driver change log 2.0.1 to 2.6.0
-

UTICK

[2.0.5]

- Improvements
 - Improved for SOC RW610.

[2.0.4]

- Bug Fixes
 - Fixed compile fail issue of no-supporting PD configuration in utick driver.

[2.0.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rules: 8.4, 14.4, 17.7

[2.0.2]

- Added new feature definition macro to enable/disable power control in drivers for some devices have no power control function.

[2.0.1]

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

[2.0.0]

- Initial version.
-

WWDT**[2.1.9]**

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

[2.1.8]

- Improvements
 - Updated the “WWDT_Init” API to add wait operation. Which can avoid the TV value read by CPU still be 0xFF (reset value) after WWDT_Init function returns.

[2.1.7]

- Bug Fixes
 - Fixed the issue that the watchdog reset event affected the system from PMC.
 - Fixed the issue of setting watchdog WDPROTECT field without considering the backwards compatibility.
 - Fixed the issue of clearing bit fields by mistake in the function of WWDT_ClearStatusFlags.

[2.1.5]

- Bug Fixes
 - deprecated a unusable API in WWDT driver.
 - * WWDT_Disable

[2.1.4]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rules Rule 10.1, 10.3, 10.4 and 11.9.
 - Fixed the issue of the inseparable process interrupted by other interrupt source.
 - * WWDT_Init

[2.1.3]

- Bug Fixes
 - Fixed legacy issue when initializing the MOD register.

[2.1.2]

- Improvements
 - Updated the “WWDT_ClearStatusFlags” API and “WWDT_GetStatusFlags” API to match QN9090. WDTOF is not set in case of WD reset. Get info from PMC instead.

[2.1.1]

- New Features
 - Added new feature definition macro for devices which have no LCOK control bit in MOD register.
 - Implemented delay/retry in WWDT driver.

[2.1.0]

- Improvements
 - Added new parameter in configuration when initializing WWDT module. This parameter, which must be set, allows the user to deliver the WWDT clock frequency.

[2.0.0]

- Initial version.
-

1.5 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.

[MCXW236B](#)

1.6 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.

1.6.1 Wireless Bluetooth LE host stack and applications

[examples__wireless_examples__bluetooth_docs](#)

1.6.2 Wireless Connectivity Framework

Wireless Framework

Wireless Connectivity Framework Connectivity Framework repository provides both connectivity platform enablement with hardware abstraction layer and a set of Services for NXP connectivity stacks : BLE, Zigbee, OpenThread, Matter.

The connectivity framework repository consists of:

- Common folder to common header files for minimal type definition to be used in the repo
- Platform folder used for platform enablement with Hardware abstraction:
 - platform/include: common API header files used by several platforms
 - platform/common: common code for several platforms
 - specifics platform folders , See below the supported platform list
 - platform/./configs folder: configuration files for framework repository and other middlewares (rpmsg, mbedTls, etc..)
- Services folder
- Zephyr folder for zephyr modules integrated in mcux SDK
- clang formatting script and script folder to format appropriately the source files of the repo

Supported platforms The following devices/platforms are supported in platform folder for connectivity applications:

- kw45x, k32w1x, mcxw71x, under wireless_mcu, kw45_k32w1_mcxw71 folders.
- kw47x, mcxw72x families under wireless_mcu, kw47_mcxw72, kw47_mcxw72_nbu folders.
- rw61x
- RT1060 and RT1170 for Matter
- Other RT devices such as i.MX RT595s

Supported services The supported services are provided for connectivity stacks and their demo application, and are usually dependent on PLATFORM API implementation:

- DBG: Light Debug Module, currently a stubbed header file
- FSCI: Framework Serial Communication Interface between BLE host stack and upper layer located on an other core/device
- FunctionLib: wrapper to toolchain memory manipulation functions (memcpy, memcmp, etc) or use its own implementation for code size reduction
- HWPParameters: Store Factory hardware parameters and Application parameters in Flash or IFR
- LowPower: wrapper of SDK power manager for connectivity applications
- ModuleInfo: Store and handle connectivity component versions
- NVM: NXP proprietary File System used for KW45, KW47 automotive devices and RT1060/RT1170 platform for Matter
- OtaSupport: Handle OTA binary writes into internal or external flash.
- SecLib and RNG: Crypto and Random Number generator functions. It supports several ports:
 - Software algorithms
 - Secure subsystem interface to an HW enclave
 - MbedTls 2.x interface
- Sensors: Provides service for Battery and temperature measurements
- SFC: Smart Frequency Calibration to be run from KW47/MCXW71 from NBU core. Matter related modules:

- OTW: Over The Wire module for External Transceiver firmware update from RT platforms
- FactoryDataProvider to be used for Matter

Supported Zephyr modules integration in mcux SDK Connectivity framework provides integration and port layers to the following Zephyr Modules located into zephyr/subsys:

- NVS: Zephyr File System used by Matter and Zigbee
- Settings: Over layer module that allows to store keys into NVS File System used by Matter Port layer and required libraries for these zephyr modules are located in port and lib folder in zephyr directory

Connectivity framework CHANGELOG

7.0.3 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_H SLOT`) 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 gHwParamsProdDataMainFlash2IfirMode_c to gHwParamsProdDataIfirMode_c
- [KW47] NBU: Add new fwk_platform_dcdc.[ch] files to allow DCDC stepping by using SPC high power mode. This requires new API in board_dcdc.c files. Please refer to new compilation MACROs gBoardDcdcRampTrim_c and gBoardDcdcEnableHighPowerModeOnNbu_d in board_platform.h files located in kw47evk, kw47loc, frdm-mcxw72 board folders.
- [KW45/MCXW71/KW47/MCXW72] Trigger an interrupt each time App core calls PLATFORM_RemoteActiveReq() to access NBU power domain in order to restart NBU core for domain low power process

Minor Changes (bug fixes)

Services

- [SecLib_RNG]
 - Rename mSecLibMutexId mutex to mSecLibSssMutexId in SecLib_sss.c
 - Remove MEM_TRACKING flag from RNG.c
 - Implement port to fsl_adapter_rng.h API using gRngUseRngAdapter_c compil Macro from RNG.c
 - Add support for BLE debug Keys in SecLi and SecLin_sss.c with gSecLibUseBleDebugKeys_d - for Debug only
- [FSCI] Add queue mechanism to prevent corruption of FSCI global variableAllow the application to override the trig sample number parameter when gFsciOverRpmsg_c is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using 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 clod 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_HciRpmsgRxCallback()
 - * Improve nbu memory allocation failure debug messages
- [SDK]
 - Multicore: remove critical section in HAL_RpmsgSendTimeout() (only required in FPGA HDI mode)
 - Flash drivers: update for ECC detection
- [Platform]
 - fwk_platform_sensors
 - * Fix temperature reporting to NBU
 - fwk_platform_extflash
 - * Align .c and .h prototype of PLATFORM_ExternalFlashAreaIsBlank() function
- [NVM]
 - Keep Mutex in NvModuleDeInit(). In Bare metal OS, Mutex can not be destroyed
 - New API NvRegisterEccFaultNotificationCb() to register Notification callback when Ecc error happens in FileSystem
- [MISRA] fixes

- SecLib_sss.c: ECDH_P256_ComputeDhKey()
- fwk_platform_extflash.c: PLATFORM_IsExternalFlashPageBlank()
- fwk_fs_abstraction.c: Various fixes
- [HWparams]
 - Fix on if condition when gHwParamsProdDataPlacementLegacy2IfrMode_c mode is selected
- [OTA]
 - Enable gOtaCheckEccFaults_d by default to avoid bus in case of ECC error during OTA
 - Fix OTA partition overflow during OTA stop and resume transfer
- [BOARD]
 - Place code button or led specific under correct defines in board_comp.c/h
 - Bring back MACROS BOARD_INITRFSWITCHCONTROL Pins in pin_mux header file of the loc board
- [SecLib]
 - Add some undefinition in SecLib_mbedtls_config as new dependency has been added in mbedtls repo:
 - * MBEDTLS_SSL_CBC_RECORD_SPLITTING, MBEDTLS_SSL_PROTO_TLS1,
 - MBEDTLS_SSL_PROTO_TLS1_1
- [FRO32K]
 - FRO32K notification callback PLATFORM_FroDebugCallback_t() has new parameter to report the fro_trim value
 - maxCalibrationIntervalMs value can be provided to NBU using PLATFORM_FwkSrvSetRfSfcConfig()
- [Sensors]
 - fix: PLATFORM_GetTemperatureValue() shall have NBU started to send temperature to NBU

6.2.1: RW61x RFP3

- [NVS]
 - Add support of NVS and Settings in framework
- [MISRA] fixes
 - board_lp.c BOARD_UninitDebugConsole() and BOARD_ReinitDebugConsole()
 - fwk_platform_ble.c: Various fixes
- [OTA]
 - Fix OTA partition overflow during OTA stop and resume transfer

6.2.0: RT1060/RT1170 SDK2.15 Major

6.1.8: KW45/K32W1 MR4

- [BOARD PLATFORM]
 - Move gBoardUseFro32k_d to board_platform.h file
 - Offer the possibility to change the source clock accuracy to gain in power consumption
- [BOARD LP]
 - Move PLATFORM_SetRamBanksRetained() at end of BOARD_EnterLowPowerCb() in case a memory allocation is done previously in this function
 - fix low power, increase BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0 to 0x10 - Skip this delay when App requesting NBU wakeup
- [PLATFORM]
 - fwk_platform_ble.c/h: New timestamp API that returns the difference between the current value of the LL clock and the argument of the function
 - fwk_platform.c/h:
 - * New PLATFORM_EnableEccFaultsAPI_d compile flag: Enable APIs for interception of ECC Fault in bus fault handler
 - * New gInterceptEccBusFaults_d compile flag: Provide FaultRecovery() demo code for bus fault handler to Intercept bus fault from Flash Ecc error
- [LOC]
 - Incorrect behavior for set_dtest_page (DqTEST11 overridden)
 - Fix SW1 button wake able on Localization board
 - Fix yellow led not properly initialized
 - Format localization pin_mux.c/h files
- [Inter Core]
 - Affect values to enumeration giving the inter core service message ids
 - Shared memory settings shared between both cores
 - Add callback to register when NBU has unrecoverable Radio issue
- [NVM]
 - Add NV_STORAGE_MAX_SECTORS, NV_STORAGE_SIZE as linker symbol for alignment with other toolchain
 - ECC detection and recovery. New gNvSalvageFromEccFault_d and gNvVerifyReadBackAfterProgram_d compile flags. Please refer to ECC Fault detection section in README.md file located in NVM folder
- [OTA]
 - Prevent bus fault in case of ECC error when reading back OTA_CFR update status (disable by default)
- [SecLib]
 - Shared mutex for RNG and SecLib as they share same hardware resource
- [Key storage]
 - Fix to ignore the garbage at the end of buffers
 - Detect when buffers are too small in KS_AddKey() functions
- [FileCache]
 - Fix deadlock in Filecache FC_Process()

- [SDK]
 - Applications: remove definition of stack location and use default from linker script, fix warmboot stack in freertos at 0x20004000
 - Memory Manager Light:
 - * fix Null pointer harfault when MEM_STATISTICS_INTERNAL enable
 - * Fix MemReinitBank() on wakeup from lowpower when Ecc banks are turned off

6.1.7: KW45/K32W1 MR3

- [OTA]
 - New API OTA_SetNewImageFlagWithOffset()
 - Fix StorageBitmapSize calculation
 - OTA clean up: Removed OTA_ValidateImage()
- [Low Power]
 - New linker Symbol m_lowpower_flag_start in linker file.
 - * Flag is used to indicate NBU that Application domain goes to power down mode. Keep this flag to 0 if only Deep sleep is supported
 - * This flag will be set to 1 if Application domain goes to power down mode
 - Re-introduce PWR_AllowDeviceToSleep()/PWR_DisallowDeviceToSleep(), PWR_IsDeviceAllowedToSleep() API
 - Implement tick compensation mechanism for idle hook in a dedicated freertos utils file fwk_freertos_utils[ch], new functions: FWK_PreIdleHookTickCompensation() and FWK_PostIdleHookTickCompensation
 - Rework timestamping on K4W1
 - * PLATFORM_GetMaxTimeStamp() based on TSTMR
 - * Rename PLATFORM_GetTimeStamp() to PLATFORM_GetTimeStamp()
 - * Update PLATFORM_Delay(): Rework to use TSTMR instead of LPTMR for platform_delay
 - * Update PLATFORM_WaitTimeout(): Fixed a bug in PLATFORM_WaitTimeout() related to timer wrap
 - * Add PLATFORM_IsTimeoutExpired() API
 - Fix race condition in PWR_EnterLowPower(), masking interrupts in case not done at upper layer
 - Low power timer split in new files fwk_platform_lowpower_timer[ch]
 - New PWR_systicks_bm.c file for bare metal usage: implement SysTick suspend/resume functionality, New weak PWR_SysTicksLowPowerInit()
- [FRO32K]
 - Improve FRO32K calibration in NBU
 - create PLATFORM_InitFro32K() to initialize FRO32K instead of XTAL32K (to be called from hardware_init())
 - update FRO32K README.md file in SFC module
 - Debug:
 - Add Notification callback feature for SFC module FRO32K

- Linker script update to support m_sfc_log_start in SMU2
- [SecLib]
 - Remove gSecLibSssUseEncryptedKeys_d compile option, split Secure/Unsecure APIs
 - RNG update to use same mutex than SecLib
 - Fix AES_128_CBC_Encrypt_And_Pad length
 - Implement RNG_ReInit() for lowpower
 - Fix issue in ECDH_P256_GenerateKeys() when waking up from power down
 - Call CRYPTO_ELEMU_reset() from SecLib_reInit() for power down support
- [BOARD]
 - Create new board_platform.h file for all Board characteristics settings (32Mhz XTAL, 32KHZ XTAL, etc..)
 - TM_EnterLowpower() TM_EnterLowpower() to be called from LP callbacks
 - Support Localization boards, Only BUTTON0 supported
 - * New compile flag BOARD_LOCALIZATION_REVISION_SUPPORT
 - * New pin_mux.[ch] files
 - Offer the possibility to override CDAC and ISEL 32MHz settings before the initialization of the crystal in board_platform.h
 - * new BOARD_32MHZ_XTAL_CDAC_VALUE, BOARD_32MHZ_XTAL_ISEL_VALUE
 - * BOARD_32MHZ_XTAL_TRIM_DEFAULT obsoleted
- [NVM file system]
 - Look ahead in pending save queue - Avoid consuming space to save outdated record
 - Fix NVM gNvDualImageSupport feature in NvIsRecordCopied
- [Inter Core]
 - Change PLATFORM_NbuApiReq() API return parameters granularity from uint32 to uint8
 - MAX_VARIANT_SZ change from 20 to 25
 - Set lp wakeup delay to 0 to reduce time of execution on host side, NBU waits XTAL to be ready before starting execution
 - Update inter core config rpmsg_config.h
 - Add timeout to while loops that relies on hardware in RemoteActiveReq(), Application can register Callbacks when timeout
 - Return non-0 status when calling PLATFORM_FwkSrvSendPacket when NBU non started
 - Let PLATFORM_GetNbuInfo return -10 if response not received on timeout - Doxygen platform_ics APIs
- [HW params]
 - New compile Macro for HW params placement in IFR - Save 8K in FLash: gHwParamsProdDataPlacement_c . 3 modes:
 - Legacy placement, move from legacy to IFR, IFR only placement
 - New compile Macro for Application data to be stored with HW params (in shared flash sector): gHwParamsAppFactoryDataExtension_d, New APIs:
 - * Nv_WriteAppFactoryData(), Nv_GetAppFactoryData()

- See HWParameter.h
- [Platform]
 - Implement PLATFORM_GetIeee802_15_4Addr() API in fwk_platform_ot.c - New gPlatformUseUniqueDeviceIdFor15_4Addr_d compile Macro
 - Wakeup NBU domain when reading RADIO_CTRL UID_LSB register in PLATFORM_GenerateNewBDAddr()
- [Reset]
 - New reset Implementations using Deep power down mode or LVD:
 - * new files fwk_platform_reset.[ch]
 - * new APIs: PLATFORM_ForceDeepPowerDownReset(), PLATFORM_ForceLvdReset() + reset on ext pins
 - * new compile flags: gAppForceDeepPowerDownResetOnResetPinDet_d and gAppForceLvdResetOnResetPinDet_d to reset on external pins
- [FSCI]
 - fix when gFsciRxAck_c enabled
 - integrate new reset APIs

6.1.4: RW610/RW612 RFP1

- [Low Power]
 - Added support of low power for OpenThread stack.
 - Added PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep/PWR_IsDeviceAllowedToSleep APIs.
- [platform]
 - Added PLATFORM_GetMaxTimeStamp API.
 - Fixed high impact Coverity.
- [FreeRTOS]
 - Created a new utilities module for FreeRTOS: fwk_freertos_utils.c/h.
 - Implemented a tick compensation mechanism to be used in FreeRTOS idle hook, likely around flash operations. This mechanism aims to estimate the number of ticks missed by FreeRTOS in case the interrupts are masked for a long time.

6.1.4: KW45/K32W1 MR2

- [Low power]
 - Powerdown mode tested and enabled on Low Power Reference Design applications
 - XTAL32K removal functionality using FRO32K, supported from NBU firmwares - limitation: Application domain supports Deep Sleep only (not power down)
 - NBU low power improvement: low power entry sequence improvement and system clock reduction to 16Mhz during WFI
 - Wake up time from cold boot, reset, power switch greatly improved. Device starts on FRO32K, switch to XTAL32K when ready if gBoardUseFro32k_d not set
 - Bug fixes:
 - * Move PWR LowPower callback to PLATFORM layers

- * Fix wrong compensation of SysTicks
- * Reinit system clocks when exiting power down mode: BOARD_ExitPowerDownCb(), restore 96MHz clock is set before going to low power
- * Call Timermanager lowpower entry exit callbacks from PLATFORM_EnterLowPower()
- * Update PLATFORM_ShutdownRadio() function to force NBU for Deep power down mode
- K32W1:
 - * Support lowpower mode for 15.4 stacks
- [NVM]
 - New Compilation MACRO gNvDualImageSupport to support multiple firmware image with different register dataset
 - Change default configuration gNvStorageIncluded_d to 1, gNvFragmentation_Enabled_d to 1, gUnmirroredFeatureSet_d to TRUE
 - Some MISRA issues for this new configuration.
 - Remove deprecated functionality gNvUseFlexNVM_d
- [SecLib]
 - New NXP Ultrafast ecp256 security library:
 - * New optimized API for ecdh DhKey/ecp256 key pair computation: Ecdh_ComputeDhKeyUltraFast(), ECP256_GenerateKeyPairUltraFast().
 - * New macro gSecLibUseDspExtension_d.
 - * Improved software version of SecLib with Ultrafast library for ECP256_LePointValid()
 - Bug fixes:
 - * Share same mutex between SecLib and RNG to prevent concurrent access to S200
 - * Optimized S200 re-initialization, restore ecdh key pair after power down
 - * Fixed race condition when power down low power entry is aborted
 - * Endianness function updates and clean up
- [OTA]
 - OTASupport improvements:
 - * New API OTA_GetImgState(), OTA_UpdateImgState()
 - * OTASupport and fwk_platform_extflash API updates for external flash: OTA_SelectExternalStoragePartition(), PLATFORM_IsExternalFlashSectorBlank(), PLATFORM_IsExternalFlashPageBlank(), PLATFORM_OtaGetOtaPartitionConfig()
 - * Updated OtaExternalFlash.c, 2 new APIs in fwk_platform_extflash.c
 - * Removed unused FLASH_op_type and FLASH_TransactionOpNode_t definitions from public API
 - * Removed unused InternalFlash_EraseBlock() from OtaInternalFlash.c
- [NBU firmware]
 - Mechanism to set frequency constraint to controller from the host PLATFORM_SetNbuConstraintFrequency()
 - NbuInfo has one more digit in versionBuildNo field

- [Board]
 - Support Extflash low power mode, add BOARD_UninitExternalFlash(), PLATFORM_UninitExternalFlash(), PLATFORM_ReinitExternalFlash()
 - Support XTAL32K removal 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

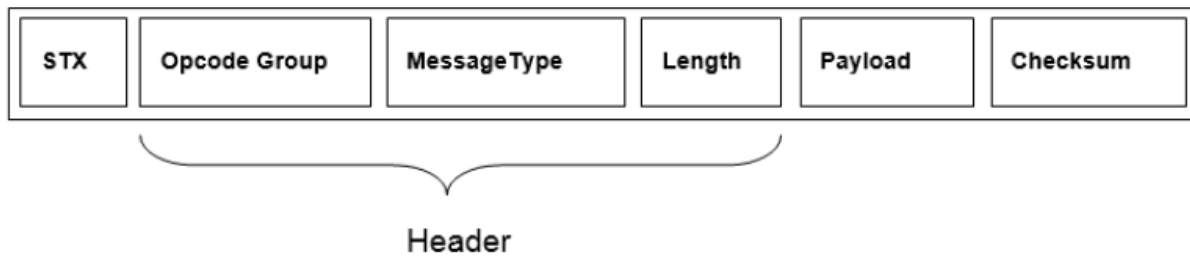
Overview The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup, the host processor (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.

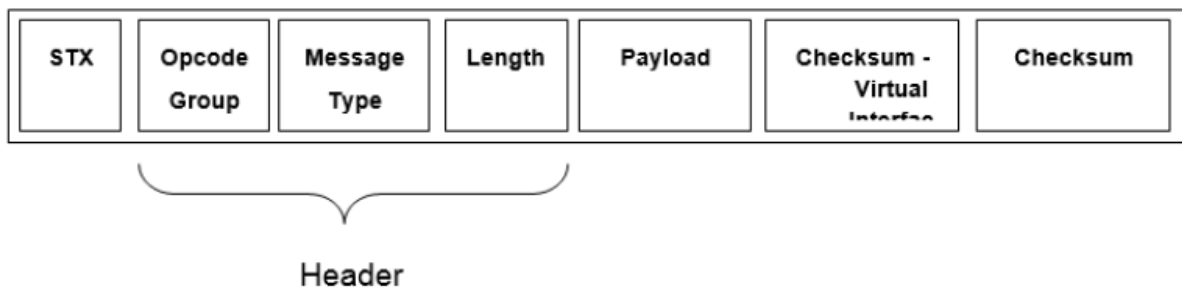
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but 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 */
```

(continues on next page)

(continued from previous page)

```
#define gFsciUseFileDataLog_c 0 /* boolean */
#define gFsciLoggingInterface_c 1 /* [0..gFsciMaxInterfaces_c) */
#define gFsciHostMacSupport_c 0 /* Host support at MAC layer */
```

The following provides the OpGroups values reserved by MAC, application, and FSCI.

FSCI Host FSCI Host is a functionality that allows separation at a certain stack layer between two entities, usually two boards running separate layers of a stack.

Support is provided for functionality at the MAC layer, for example, MAC/PHY layers of a stack are running as a Black Box on a board, and MAC higher layers are running on another. The higher layers send and receive serial commands to and from the MAC Black Box using the FSCI set of operation codes and groups.

The protocol of communication between the two is the same. The current level of support is provided for:

- FSCI_MsgResetCPUReqFunc – sends a CPU reset request to black box
- FSCI_MsgWriteExtendedAdrReqFunc – configures MAC extended address to the Black Box
- FSCI_MsgReadExtendedAdrReqFunc – N/A

The approach on the Host interfacing a Black Box using synchronous primitives is by default the polling of the FSCI_receivePacket function, until the response is received from the Black Box. The calling task polls whenever the task is being scheduled. This is required because a stack synchronous primitive requires that the response of that request is available in the context of the caller right after the SAP call has been executed.

The other option, available for RTOS environments, is using an event mechanism. The calling task blocks waiting for the event that is sent from the Serial Manager task when the response is available from the Black Box. This option is disabled by default. The disadvantage of this option is that the primitive cannot be received from another Black Box through a serial interface because the blocked task is the Serial Manager task, which reaches a deadlock as cannot be released again.

FSCI ACK ACK transmission is enabled through the gFsciTxAck_c macro definition. Each FSCI valid packet received triggers an FSCI ACK packet transmission on the same FSCI interface that the packet was received on. The serial write call is performed synchronously to send the ACK packet before any other FSCI packet. Only then the registered handler is called to process the received packet. The ACK is represented by the gFSCI_CnfOpcodeGroup_c and mFsciMsgAck_c Opcode. An additional byte is left empty in the payload so that it can be used optionally as a packet identifier to correlate packets and ACKs. ACK reception is the other component that is enabled through gFsciRxAck_c. The behavior is such that every FSCI packet sent through a serial interface triggers an FSCI ACK packet reception on the same interface after the packet is sent. If an ACK packet is received, the transmission is considered successful. Otherwise, the packet is resent a number of times. The ACK wait period is configurable through mFsciRxAckTimeoutMs_c and the number of transmission retries through mFsciTxRetryCnt_c. The ACK mechanism described above can also be coupled with a FSCI packet reception timeout enabled through gFsciRxTimeout_c and configurable through mFsciRxRestartTimeoutMs_c. Whenever there are no more bytes to be read from a serial interface, a timeout is configured at the predefined value if no other bytes are received. If new bytes are received, the timer is stopped and eventually canceled at successful reception. However, if, for any reason, the timeout is triggered, the FSCI module considers that the current packet is invalid, drops it, and searches for a new start marker.

FSCI usage example Detailed data types and APIs are described in ConnFWK API documentation.

Initialization

```
/* Configure the number of interfaces and virtual interfaces used */
#define gFsciMaxInterfaces_c 4
#define gFsciMaxVirtualInterfaces_c 2
....
/* Define the interfaces used */
static const gFsciSerialConfig_t myFsciSerials[] = {
    /* Baudrate, interface type, channel No, virtual interface */ {gUARTBaudRate115200_c, gSerialMgrUart_
↪c, 1, 0}, {gUARTBaudRate115200_c, gSerialMgrUart_c, 1, 1}, {0 , gSerialMgrIICSlave_c, 1, 0}, {0 ,
↪gSerialMgrUSB_c, 0, 0},
};
....
/* Call init function to open all interfaces */
FSCI_Init( (void*)myFsciSerials );
```

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 */
```

(continues on next page)

(continued from previous page)

```
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 */
}
```

(continues on next page)

(continued from previous page)

```

↪ */
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) gHwParamsProdDataMainFlash2IfrMode_c(1): - PROD_DATA are located in IFR1, but MainFlash version still exists during interim period. - If the contents of the PROD_DATA section in MainFlash is valid (not blank and correct CRC) but the IFR PROD_DATA is still blank, copy the contents of MainFlash PROD_DATA to IFR location. - When done PROD_DATA in IFR are used. Once the transition is done, an application using (2: gHwParamsProdDataPlacementIfrMode_c) may be programmed.
 - 3) gHwParamsProdDataIfrMode_c (2) :
 - PROD_DATA section dwells in the IFR1 sector [0x02002000..0x02004000[
 - in development phase the area comprised between [0x02002000..0x02002600[must be reserved for internal purposes.
 - This allows to free up the top sector of Main Flash by linking with gUseProdInfoMainFlash_d unset.

LowPower

Low Power reference user guide This Readme file describes the connectivity software architecture and provides the general low power enablement user guide.

1- Connectivity Low Power SW architecture The connectivity low power software architecture is composed of various components. These are described from the lower layer to the application layer:

1. The SDK power manager in component/power_manager. This component provides the basic low power framework. It is not specific to the connectivity but generic across devices. it covers:
 - gather the low power constraints for upper layer and take the decision on the best suitable low power state the device is allowed to go to fulfill 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.

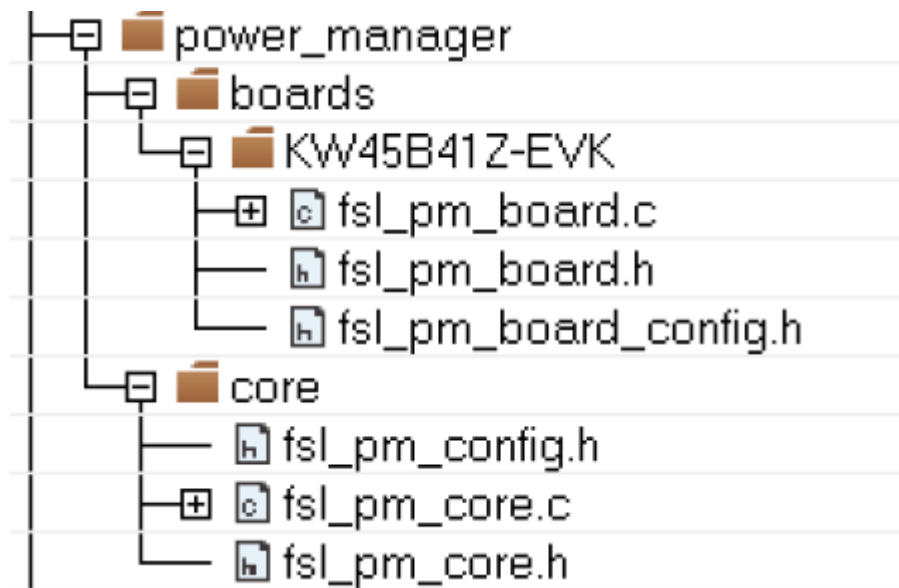
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

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.

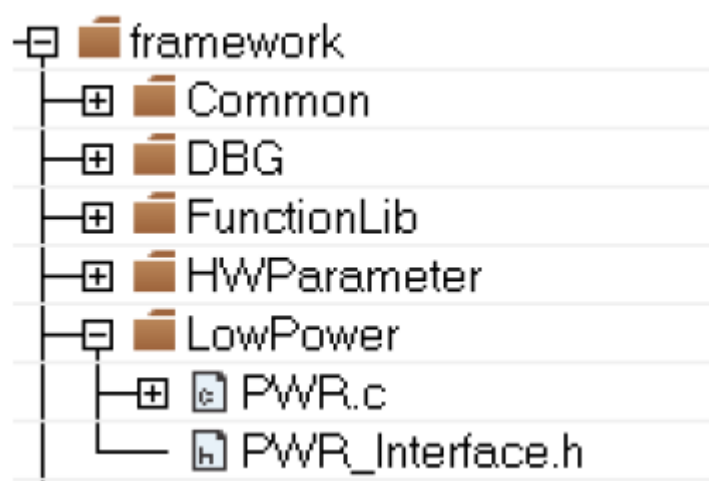
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

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.

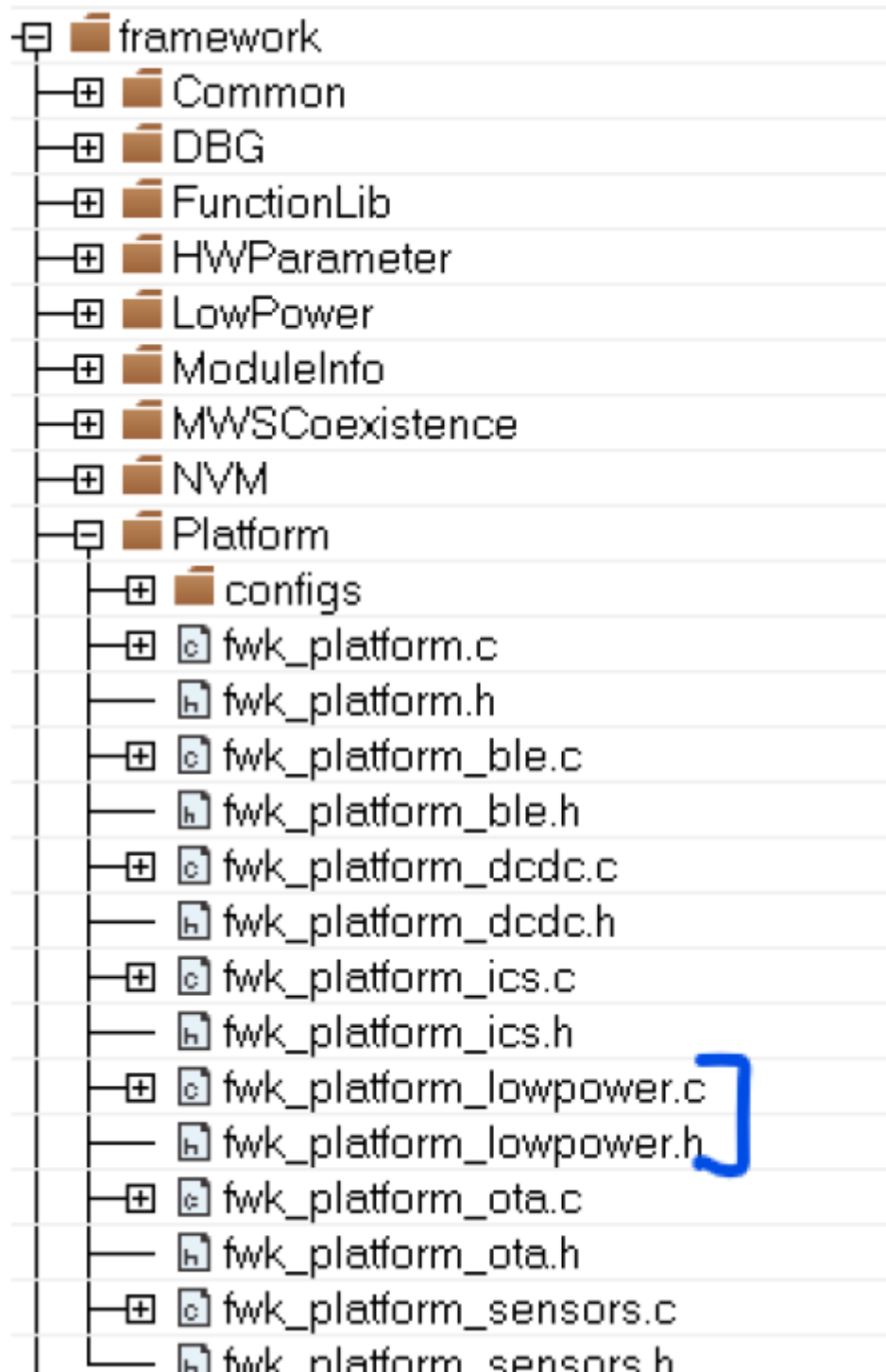
2.1.2 - PWR connectivity framework module PWR.c PWR_Interface.h shall be added to your application projects :



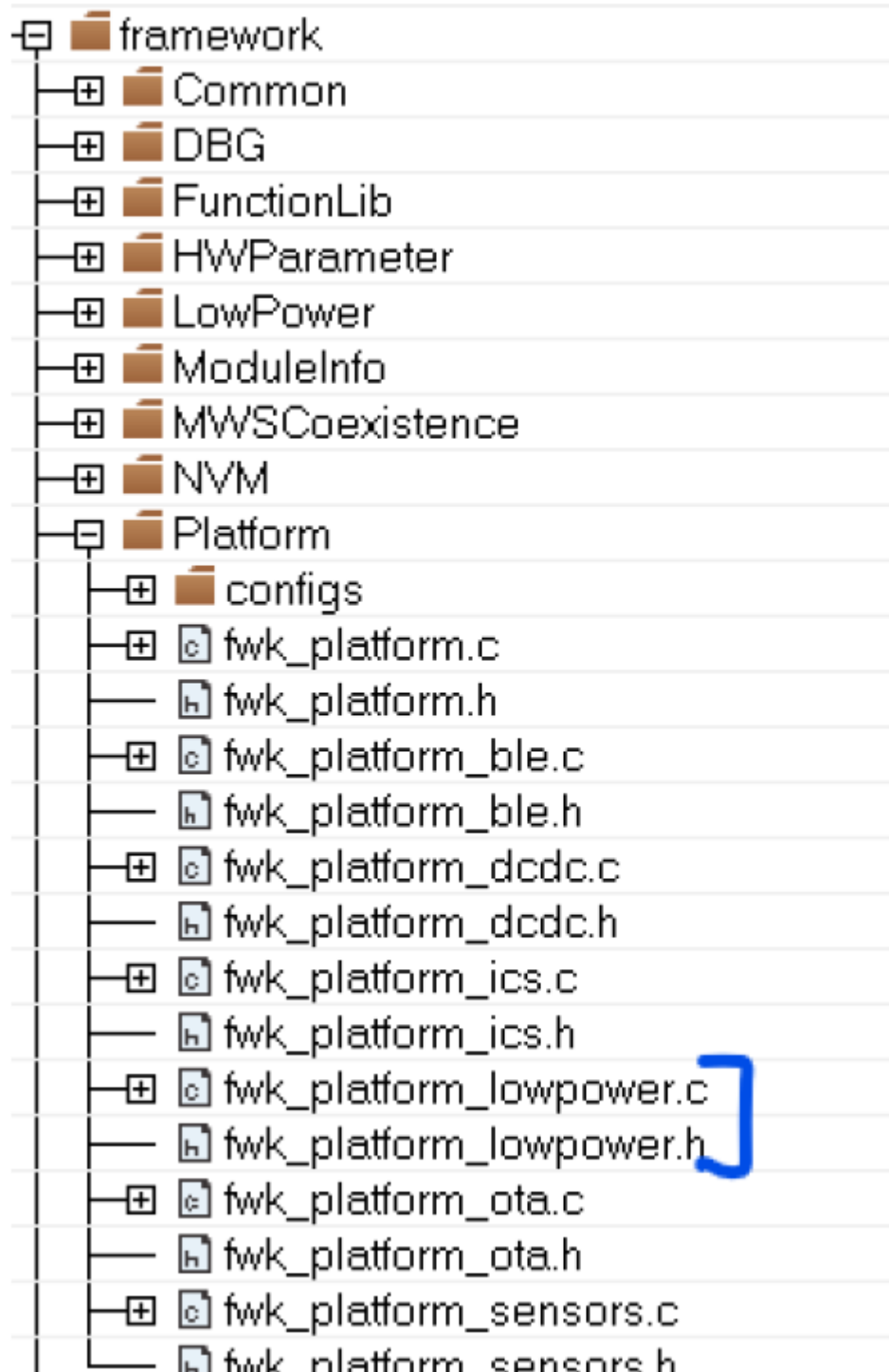
Optionally, in order to support SysTick less mode, `PWR_systicks.c` or `PWR_systicks_bm.c` could also be added.

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

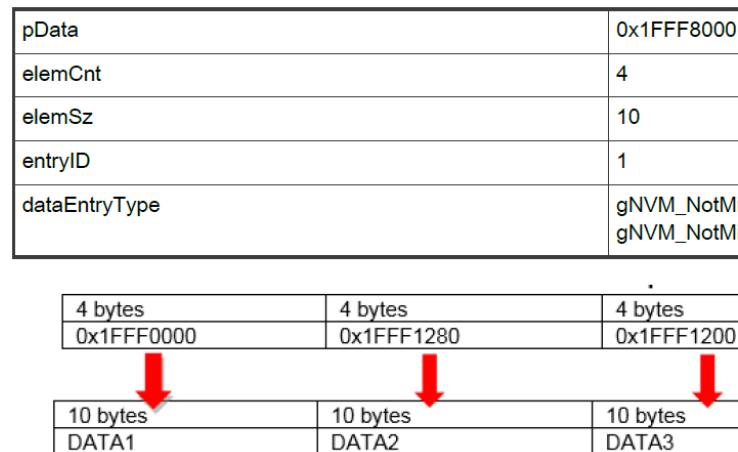
NVM Table entry As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

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

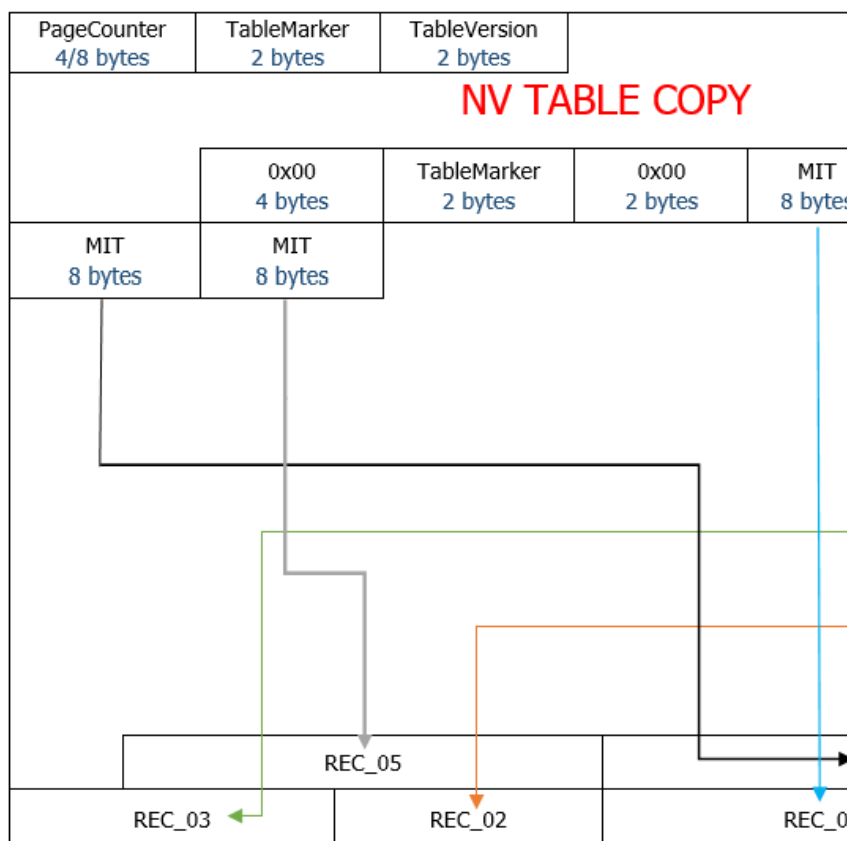
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page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

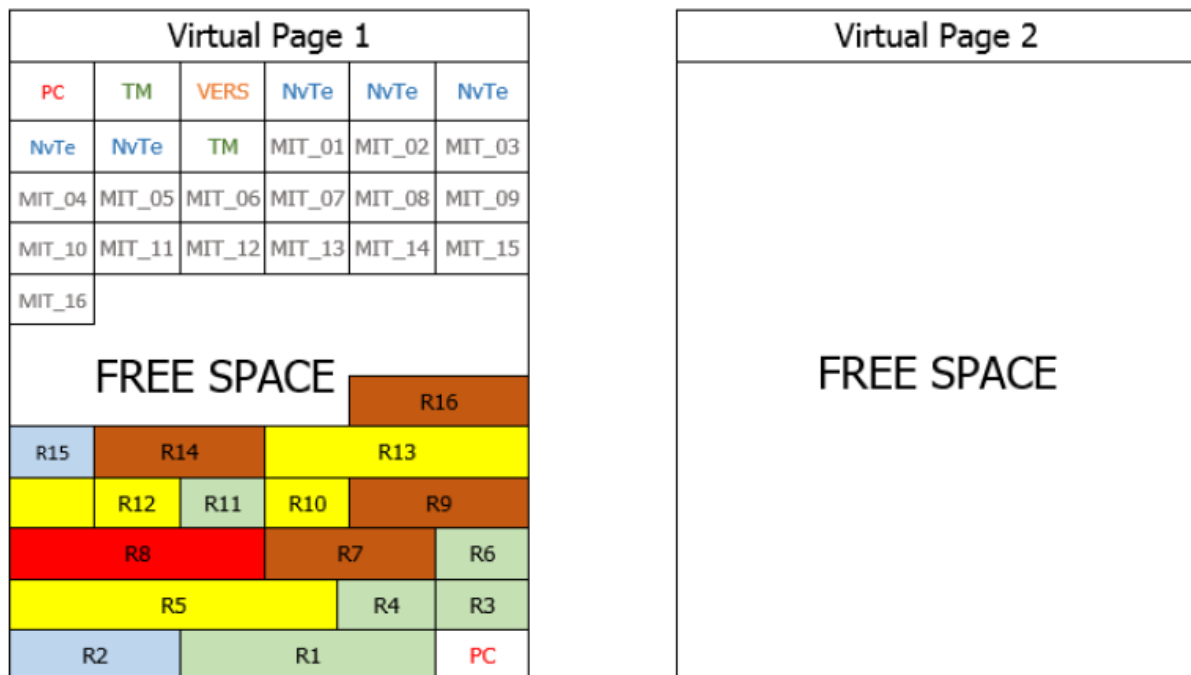
VSB	entryID	elemIdx	recordOffset	VEB
1 byte	2 bytes	2 bytes	2 bytes	

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.

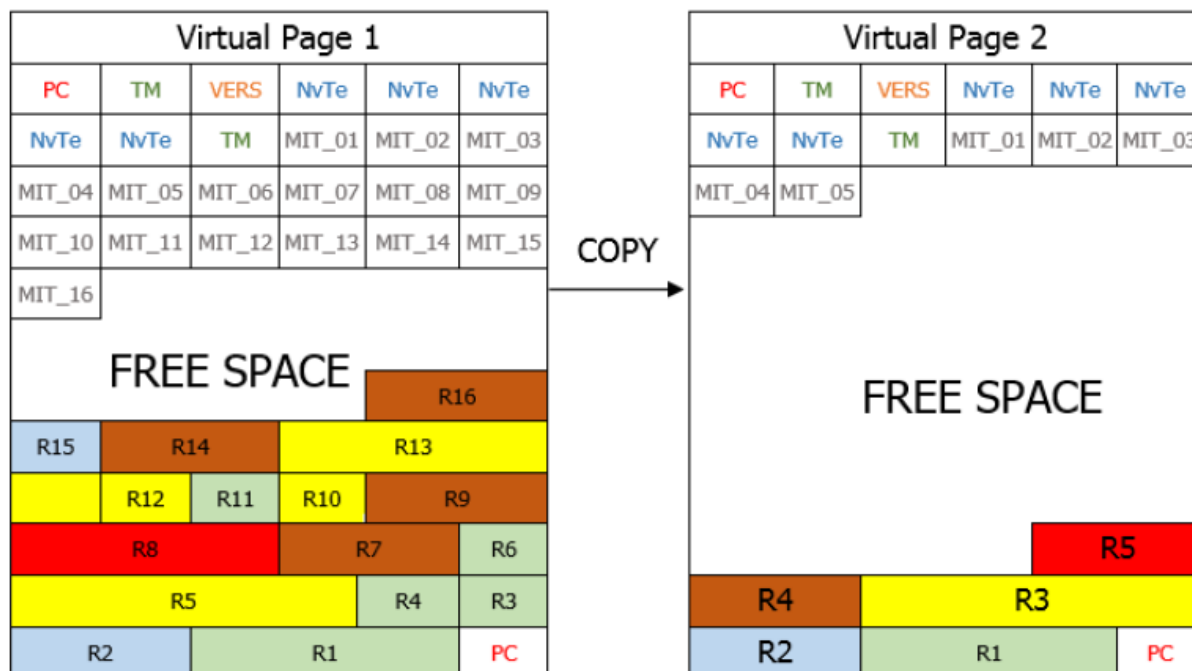


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- 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 0xFFFFU.
- *gNvFormatRetryCount_c* : Macro used to define the maximum retries count value for the format operation. It is set to 3 by default.
- *gNvPendingSavesQueueSize_c* : Macro used to define the size of the pending saves queue. It is set to 32 by default.
- *gFifoOverwriteEnabled_c* : Macro used to enable overwriting older entries in the pending saves queue (if it is full). If it is FALSE and the queue is full, the module tries to process the oldest save in the queue to free a position. It is set to FALSE by default.
- *gNvMinimumFreeBytesCountStart_c* : Macro used to define the minimum free space at initialization. If the free space is smaller than this value, a page copy is triggered. It is set by default to 128.
- *gNvEndOfTableId_c* : Macro used to define the ID of the end-of-table entry. It is set to 0xFFFEU by default. No valid entry should use this ID.
- *gNvTableMarker_c* : Macro used to define the table marker value. The table marker is used to indicate the start and the end of the flash copy of the NV table. It is set to 0x4254U by default.
- *gNvFlashTableVersion_c* : Macro used to define the flash table version. It is used to determine if the NVM table was updated. It is set to 1 by default. The application should modify this every time the NVM table is updated and the data from NVM is still required.
- *gNvTableKeptInRam_d* : Set *gNvTableKeptInRam_d* to FALSE, if the NVM table is stored in FLASH memory (default). If the NVM table is stored in RAM memory, set the macro to TRUE.
- *gNvVerifyReadBackAfterProgram_d* : set by default force verification of NVM programming operations. Is forced implicitly when *gNvSalvageFromEccFault_d* is defined.
- *gNvSalvageFromEccFault_d* : use safe flash API to read from flash, and provide corrective action when ECC fault is met.

OtaSupport: Over-the-Air Programming Support

Overview This module includes APIs for the over-the-air image upgrade process. A Server device receives an image over the serial interface from a PC or other device thorough FSCI commands. If the Server has an image storage, the image is saved locally. If not, the image is requested chunk by chunk: With image storage

- *OTA_RegisterToFsci()*
- *OTA_InitExternalMemory()*
- *OTA_WriteExternalMemory()*
- ...
- *OTA_WriteExternalMemory()*

Without image storage:

- *OTA_RegisterToFsci()*
- *OTA_QueryImageReq()*
- *OTA_ImageChunkReq()*
- ...
- *OTA_ImageChunkReq()*

A Client device processes the received image by computing the CRC and filter unused data and stores the received image into a non-volatile storage. After the entire image has been transferred and verified, the Client device informs the Bootloader application that a new image is available, and that the MCU must be reset to start the upgrade process. See the following command sequence:

- `OTA_StartImage()`
- `OTA_PushImageChunk()` and `OTA_CrcCompute ()`
- ...
- `OTA_PushImageChunk()` and `OTA_CrcCompute ()`
- `OTA_CommitImage()`
- `OTA_SetNewImageFlag()`
- `ResetMCU()`

SecLib_RNG: Security library and random number generator

Random number generator

Overview The RNG module is part of the framework used for random number generation. It uses hardware RNG peripherals as entropy sources (TRNG, Secure Subsystem, ...) to provide a true random number generator interface. A Pseudo-Random number generator (PRNG) implementation is available. The PRNG may depend of SecLib services (thus requiring a common mutex) to perform HMAC-SHA256, SHA256, AES-CTR, or alternatively a Lehmer Linear Congruential generator. A prerequisite for the PRNG to function with desired randomness is to be seeded using a proper source of entropy. If no hardware acceleration is present, the RNG may fallback to lesser quality ad-hoc source e.g if present `SIM_UID` registers, the UIDL is used as the initial seed for the random number generator.

Initialization The RNG module requires an initialization via a call to `RNG_Init`. The RNG initialization involves a call to `RNG_SetSeed`.

In the case of a dual core system consisting of a Host core and an NBU core, the Secure Subsystem is owned by the Host core. The Host core then has a direct access to its TRNG embedded in its secure subsystem. On the NBU code side, a request is emitted via `RPMSG` to the Host to provide a seed. On receipt of this request, the Host sets a 'reseed needed' flag (from the ISR context). If the core running the RNG service owns the TRNG entropy hardware (if any), it can get the seed directly from this hardware synchronously. In the case of an NBU that does not control the device's entropy source, that is owned by the Host, it requests a seed from the Host processor via `RPMSG` exchange. On receipt of this request the Host sets a flag notifying of this request from the `RPMSG` ISR context. From the Idle thread, this flag is polled via the `RNG_IsReseedNeeded` API. If set the seed is regenerated and forwarded to the NBU via `RPMSG`.

`RNG_ReInit` API is to be used at wake up time in the context of `LowPower`. `RNG_DeInit` is used for unit tests and coverage purposes but has no useful role in a real application.

Seed handling `RNG_SetSeed`: `RNG_SetExternalSeed` may be used to inject application entropy to RNG context seed using a supplied array of bytes. `RNG_IsReseedNeeded` used from task in Host core to check whether seed must be sent to NBU core.

`RNG_GetTrueRandomNumber` is the API used to generate a Random 32 bit number from a HW source of entropy. It is essential if only to seed the pseudo random number generator.

`RNG_GetPseudoRandomData` is used to generate arrays of random bytes.

Security Library

Overview The framework provides support for cryptography in the security module. It supports both software and hardware encryption. Depending on the device, the hardware encryption uses either the S200, MMCAU, LTC, or CAU3 module instruction set or dedicated AES and SHA hardware blocks.

Software implementation is provided in a library format.

Support for security algorithms

		SW SecLib : SecLib.c	EdgeLock SecLib_sss.c	SecLib_ecdh.c	Mbedtls SecLib_mbedtls	nccl (part of SecLib.c)	Usage example
AES_128		SecLib_aes.c	x		x		
AES_128_ECB			x		x		
AES_128_CBC		x	x		x		
AES_128_CTR encryption	en-	x	x				
AES_128_OFB encryption	En-	x					
AES_128_CMAC		x	x		x		BLE connection, ieee 15.4
AES_128_EAX		x					
AES_128_CCM		x	x		x		BLE, ieee 15.4
SHA1		SecLib_sha.c	x		x		
SHA256		x	x		x		
HMAC_SHA256		x	x		x		PRNG, Digest for Matter
ECDH_P256 shared secret generation		x (by 15 incremental steps) -> SecLib_ecdh.c	x with MACRO SecLibECDHUseSSS	x	x	x	BLE pairing,
EC_P256 key pair generation		x	x	x	x	x	
EC_P256 public key generation from private key				x	x	x	Matter (ECDSA)
ECDSA_P256 hash and msg signature generation / verification			only if owner of the key pair		x	x	Matter
SPAKE2+ P256 arithmetics					x	x	Matter

BLE advanced secure mode

New elements in existing structures: `computeDhKeyParam_t::keepInternalBlob` - boolean telling if the shared blob is kept in this structure(in `.outpoint`) after `ECDH_P256_ComputeDhKey()` or `ECDH_P256_ComputeDhKeySeg()` call.

New arguments in existing functions: `ECDH_P256_ComputeDhKey` `keepBlobDhKey` - boolean telling `ECDH_P256_ComputeDhKey()` or `ECDH_P256_ComputeDhKeySeg()` to keep the shared object after computation for later use (it is required by the `SecLib_GenerateBluetoothF5KeysSecure`).

New macros: `gSecLibSssUseEncryptedKeys_d` - Enable or disable S200 blobs SecLib support. 0 - the Bluetooth Keys are available in plaintext, 1 - the Bluetooth Keys are not available in plaintext, but in secured blobs. Default is disabled.

New functions:

LE Secure connections pairing:

`void ECDH_P256_FreeDhKeyDataSecure` This is a function used to free the shared object stored in `computeDhKeyParam_t`. When user calls `ECDH_P256_ComputeDhKeySeg()` with `keepBlobDhKey` set to 1, it should also call **`ECDH_P256_FreeDhKeyDataSecure`** .

`SecLib_GenerateBluetoothF5Keys` This function is extracted from the Bluetooth LE Host Stack implementation. This corresponds to the legacy implementation without key blobs.

`SecLib_GenerateBluetoothF5KeysSecure` Similar to **`SecLib_GenerateBluetoothF5Keys`** this function is modified to work with key blobs, the reason is to not use SSS inside the Bluetooth LE Host Stack.

`SecLib_DeriveBluetoothSKD` This is a helper function used by the Bluetooth LE Host Stack in the pairing procedure, when receiving the vendor HCI command specifying that the ESK needs to be provided to LL.

`ELKE_BLE_SM_F5_DeriveKeys` This is a private function, helper for **`SecLib_GenerateBluetoothF5KeysSecure`**. It was provided by the STEC team.

Privacy:

`SecLib_ObfuscateKeySecure` This is a function used by the Bluetooth LE Host Stack to obfuscate the IRK before setting it to Bluetooth LE Controller or before saving it to NVM

`SecLib_DeobfuscateKeySecure` This is a function used by the Bluetooth LE Host Stack to extract the plaintext IRK key from the saved NVM blob.

SecLib_VerifyBluetoothAh This function is extracted from the legacy Bluetooth LE Host Stack implementation using plaintext keys.

SecLib_VerifyBluetoothAhSecure Similar to **SecLib_VerifyBluetoothAh** with modification to work with S200 key blob.

SecLib_GenerateSymmetricKey This is a function used by the application to generate the local IRK and local CSRK.

SecLib_GenerateBluetoothEIRKBlobSecure This is a function used by the application to generate the EIRK needed by Bluetooth LE Controller from the IRK blob.

A2B feature

ECDH_P256_ComputeA2BKey This function is used to compute the EdgeLock to EdgeLock key. pInPeerPublicKey points to the peer public key, pOutE2EKey is the pointer to where the E2E key object will be stored, this will be freed by the application when it is no longer required by calling ECDH_P256_FreeE2EKeyData().

ECDH_P256_FreeE2EKeyData This function is used to free the key object given as a parameter. It is used by the application to free the E2E key when is no longer needed.

SecLib_ExportA2BBlobSecure This function is used to import an ELKE blob or plain text symmetric key in s200 and export an E2E key blob. The input type is identified by the keyType parameter.

SecLib_ImportA2BBlobSecure This function is used to import an E2E key blob in s200 and export an ELKE blob or plain text symmetric key. The output type is identified by the keyType parameter.

LE Secure connections Pairing flow and SecLib usage:

1. Each device needs to generate locally the public+private keypair. This is done using **ECDH_P256_GenerateKeys**.
2. Devices exchange their public keys.
3. Upon receiving the peer device's public key, local device is computing DH key using **ECDH_P256_ComputeDhKey**.
4. Each device sends DHKeyCheck packet
5. Upon receiving DhKeyCheck each device computes LTK blob using **SecLib_GenerateBluetoothF5Keys**
6. After computing the each device sends HCI_LeStartEnc (on initiator), HCI_Le_Provide_Long_Term_Key (on responder)
7. Bluetooth LE Controller sends back SKD report custom event
8. Bluetooth LE Host Stack computes ESKD based on LTK blob using **SecLib_DeriveBluetoothSKD** and sends it to Bluetooth LE Controller
9. Bluetooth LE Controller encrypts the link

IRK flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received:
 - the local IRK is generated using **SecLib_GenerateSymmetricKey**
 - the local EIRK is generated using **SecLib_GenerateBluetoothEIRKBlobSecure**
 - local CSRK is generated using **SecLib_GenerateSymmetricKey**
2. During legacy pairing when receiving bonding keys, IRK is obfuscated using **SecLib_ObfuscateKeySecure** and stored
3. When app wants to set the OOB keys using Gap_SaveKeys the IRK is obfuscated using **SecLib_ObfuscateKeySecure**
4. When application calls API Gap_VerifyPrivateResolvableAddress IRK is obfuscated using **SecLib_ObfuscateKeySecure** and verified using **SecLib_VerifyBluetoothAhSecure**
5. When a new connection is received in Host with RPA address not resolved by the Bluetooth LE Controller, the Host tries to resolve it by obfuscating it using **SecLib_ObfuscateKeySecure** and verifying it using **SecLib_VerifyBluetoothAhSecure**
6. When adding a peer in Bluetooth LE Controller resolving list, the peer's IRK is obfuscated using **SecLib_ObfuscateKeySecure** before setting it using **HCI_Le_Add_Device_To_Resolving_List**.
7. When an IRK plaintext is requested by the application using Gap_LoadKeys it is obtained using **SecLib_DeobfuscateKeySecure**
8. When legacy pairing completes and LTK needs to be send in the pairing complete event (gConnEvtPairingComplete_c) the **SecLib_DeobfuscateKey** is used to extract the plaintext.

A2B flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received, the application will call **ECDH_P256_GenerateKeys** to generate the public/private key pair required for the E2E key derivation and send the public key to the peer device.
2. When the public key is received from the peer device, the application will call **ECDH_P256_ComputeA2BKeySecure** to generate the EdgeLock to EdgeLock key.
3. The application will obtain an E2E IRK blob by calling **SecLib_ExportA2BBlobSecure** with key type gSecElkeBlob_c. The obtained blob is sent to the peer anchor. The peer anchor will call **SecLib_ImportA2BBlob** with keyType gSecElkeBlob_c and save the resulting ELKE blob in NVM, for Digital Key both anchors must have the same IRK.
4. After pairing, in order to send the LTK and IRK contained in the bonding data securely, the application will call **SecLib_ExportA2BBlobSecure** with keyType gSecLtkElkeBlob_c for the LTK, and **SecLib_ExportA2BBlobSecure** with keyType gSecPlainText_c for the IRK. The E2E blobs obtained are sent along with the rest of the bonding data to the peer anchor device.
5. After the bonding data is trasfered the E2E key is no longer needed and **ECDH_P256_FreeE2EKeyData** is called with the key object obtained at step 2 when **ECDH_P256_ComputeA2BKeySecure** was called.

Sensors

Overview The Sensors module provides an API to communicate with the ADC. Two values can be obtained by this module :

- Temperature value

- Battery level

The temperature is given in tenths of degrees Celsius and the battery in percentage.

This module is multi-caller, the ADC is protected by a mutex on the resource and by preventing lowpower (only WFI) during its processing. Platform specific code can be find in `fwk_platform_sensors.c/h`.

Constant macro definitions Name :

```
#define VALUE_NOT_AVAILABLE_8 0xFFu
#define VALUE_NOT_AVAILABLE_32 0xFFFFFFFFu
```

Description :

Defines the error value that can be compared to the value obtain on the ADC.

SFC : Smart Frequency Calibration

Overview The Smart Frequency Calibration module provides operations and calibration for the FRO32K source clock. This module is split between main core and Radio core:

- `fwk_rf_sfc.[ch]`: RF_SFC module on Radio core that provides Main FRO32K measurement/calibration and state machine in synchronization 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 swith 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.

1.6.3 FreeRTOS

FreeRTOS

Chapter 2

MCXW236B

2.1 ANACTRL: Analog Control Driver

`void ANACTRL_Init(ANACTRL_Type *base)`

Initializes the ANACTRL mode, the module's clock will be enabled by invoking this function.

Parameters

- `base` – ANACTRL peripheral base address.

`void ANACTRL_Deinit(ANACTRL_Type *base)`

De-initializes ANACTRL module, the module's clock will be disabled by invoking this function.

Parameters

- `base` – ANACTRL peripheral base address.

`void ANACTRL_SetFro192M(ANACTRL_Type *base, const anactrl_fro192M_config_t *config)`

Configures the on-chip high-speed Free Running Oscillator(FRO192M), such as enabling/disabling 12 MHz clock output and enable/disable 96MHz clock output.

Parameters

- `base` – ANACTRL peripheral base address.
- `config` – Pointer to FRO192M configuration structure. Refer to `anactrl_fro192M_config_t` structure.

`void ANACTRL_GetDefaultFro192MConfig(anactrl_fro192M_config_t *config)`

Gets the default configuration of FRO192M. The default values are:

```
config->enable12MHzClk = true;  
config->enable96MHzClk = false;
```

Parameters

- `config` – Pointer to FRO192M configuration structure. Refer to `anactrl_fro192M_config_t` structure.

`void ANACTRL_SetXo32M(ANACTRL_Type *base, const anactrl_xo32M_config_t *config)`

Configures the 32 MHz Crystal oscillator(High-speed crystal oscillator), such as enable/disable output to CPU system, and so on.

Parameters

- `base` – ANACTRL peripheral base address.
- `config` – Pointer to XO32M configuration structure. Refer to `anactrl_xo32M_config_t` structure.

`void ANACTRL_GetDefaultXo32MConfig(anactrl_xo32M_config_t *config)`

Gets the default configuration of XO32M. The default values are:

```
config->enableSysClkOutput = false;  
config->enableACBufferBypass = false;
```

Parameters

- `config` – Pointer to XO32M configuration structure. Refer to `anactrl_xo32M_config_t` structure.

`uint32_t ANACTRL_MeasureFrequency(ANACTRL_Type *base, uint8_t scale, uint32_t refClkFreq)`

Measures the frequency of the target clock source.

This function measures target frequency according to a accurate reference frequency. The formula is: $F_{target} = (CAPVAL * F_{reference}) / ((1 \ll SCALE) - 1)$

Note: Both target and reference clocks are selectable by programming the target clock select `FREQMEAS_TARGET` register in `INPUTMUX` and reference clock select `FREQMEAS_REF` register in `INPUTMUX`.

Parameters

- `base` – ANACTRL peripheral base address.
- `scale` – Define the power of 2 count that ref counter counts to during measurement, ranges from 2 to 31.
- `refClkFreq` – frequency of the reference clock.

Returns

frequency of the target clock.

`static inline void ANACTRL_EnableInterrupts(ANACTRL_Type *base, uint32_t mask)`

Enables the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

`static inline void ANACTRL_DisableInterrupts(ANACTRL_Type *base, uint32_t mask)`

Disables the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

`static inline void ANACTRL_ClearInterrupts(ANACTRL_Type *base, uint32_t mask)`

Clears the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

static inline uint32_t ANACTRL_GetStatusFlags(ANACTRL_Type *base)

Gets ANACTRL status flags.

This function gets Analog control status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_flags`. For example, to check whether the flash is in power down mode:

```
if (kANACTRL_FlashPowerDownFlag & ANACTRL_ANACTRL_GetStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL status flags which are given in the enumerators in the `_anactrl_flags`.

static inline uint32_t ANACTRL_GetOscStatusFlags(ANACTRL_Type *base)

Gets ANACTRL oscillators status flags.

This function gets Anactrl oscillators status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_osc_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_osc_flags`. For example, to check whether the FRO192M clock output is valid:

```
if (kANACTRL_OutputClkValidFlag & ANACTRL_ANACTRL_GetOscStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the `_anactrl_osc_flags`.

static inline uint32_t ANACTRL_GetInterruptStatusFlags(ANACTRL_Type *base)

Gets ANACTRL interrupt status flags.

This function gets Anactrl interrupt status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_interrupt_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_interrupt_flags`. For example, to check whether the VBAT voltage level is above the threshold:

```
if (kANACTRL_BodVbatPowerFlag & ANACTRL_ANACTRL_GetInterruptStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the `_anactrl_osc_flags`.

static inline void ANACTRL_EnableVref1V(ANACTRL_Type *base, bool enable)

Aux_Bias Control Interfaces.

Enables/disables 1V reference voltage buffer.

Parameters

- base – ANACTRL peripheral base address.
- enable – Used to enable or disable 1V reference voltage buffer.

enum __anactrl_interrupt_flags

ANACTRL interrupt flags.

Values:

enumerator kANACTRL_Bod1Flag

BOD1 Interrupt status before Interrupt Enable.

enumerator kANACTRL_Bod1InterruptFlag

BOD1 Interrupt status after Interrupt Enable.

enumerator kANACTRL_Bod1PowerFlag

Current value of BOD1 power status output.

enumerator kANACTRL_BodCoreFlag

BOD CORE Interrupt status before Interrupt Enable.

enumerator kANACTRL_BodCoreInterruptFlag

BOD CORE Interrupt status after Interrupt Enable.

enumerator kANACTRL_BodCorePowerFlag

Current value of BOD CORE power status output.

enumerator kANACTRL_DcdcFlag

DCDC Interrupt status before Interrupt Enable.

enumerator kANACTRL_DcdcInterruptFlag

DCDC Interrupt status after Interrupt Enable.

enumerator kANACTRL_DcdcPowerFlag

Current value of DCDC power status output.

enum __anactrl_interrupt

ANACTRL interrupt control.

Values:

enumerator kANACTRL_Bod1InterruptEnable

BOD1 interrupt control.

enumerator kANACTRL_BodCoreInterruptEnable

BOD CORE interrupt control.

enumerator kANACTRL_DcdcInterruptEnable

DCDC interrupt control.

enum __anactrl_flags

ANACTRL status flags.

Values:

enumerator kANACTRL_FlashPowerDownFlag

Flash power-down status.

enumerator kANACTRL_FlashInitErrorFlag
Flash initialization error status.

enum _anactrl_osc_flags
ANACTRL FRO192M and XO32M status flags.

Values:

enumerator kANACTRL_OutputClkValidFlag
Output clock valid signal.

enumerator kANACTRL_CCOTresholdVoltageFlag
CCO threshold voltage detector output (signal vcco_ok).

enumerator kANACTRL_XO32MOutputReadyFlag
Indicates XO out frequency stability.

typedef struct *_anactrl_fro192M_config* anactrl_fro192M_config_t
Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

typedef struct *_anactrl_xo32M_config* anactrl_xo32M_config_t
Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

FSL_ANACTRL_DRIVER_VERSION
ANACTRL driver version.

struct _anactrl_fro192M_config
#include <fsl_anactrl.h> Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

Public Members

bool enable12MHzClk
Enable 12MHz clock.

bool enable96MHzClk
Enable 96MHz clock.

struct _anactrl_xo32M_config
#include <fsl_anactrl.h> Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

Public Members

`bool enableACBufferBypass`

Enable XO AC buffer bypass in pll and top level.

`bool enableSysCLkOutput`

Enable XO 32 MHz output to CPU system, SCT, and CLKOUT

`bool enableADCOutput`

Enable High speed crystal oscillator output to ADC.

2.2 CASPER: The Cryptographic Accelerator and Signal Processing Engine with RAM sharing

2.3 casper_driver

`FSL_CASPER_DRIVER_VERSION`

CASPER driver version. Version 2.2.4.

Current version: 2.2.4

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Bug fix KPSDK-24531 `double_scalar_multiplication()` result may be all zeroes for some specific input
- Version 2.0.2
 - Bug fix KPSDK-25015 `CASPER_MEMCPY` hard-fault on LPC55xx when both source and destination buffers are outside of `CASPER_RAM`
- Version 2.0.3
 - Bug fix KPSDK-28107 `RSUB`, `FILL` and `ZERO` operations not implemented in `enum_casper_operation`.
- Version 2.0.4
 - For GCC compiler, enforce `O1` optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires `-fno-strict-aliasing`.
- Version 2.0.5
 - Fix sign-compare warning.
- Version 2.0.6
 - Fix IAR Pa082 warning.
- Version 2.0.7
 - Fix MISRA-C 2012 issue.
- Version 2.0.8
 - Add feature macro for `CASPER_RAM_OFFSET`.
- Version 2.0.9
 - Remove unused function `Jac_oncurve()`.
 - Fix ECC384 build.

- Version 2.0.10
 - Fix MISRA-C 2012 issue.
- Version 2.1.0
 - Add ECC NIST P-521 elliptic curve.
- Version 2.2.0
 - Rework driver to support multiple curves at once.
- Version 2.2.1
 - Fix MISRA-C 2012 issue.
- Version 2.2.2
 - Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE
- Version 2.2.3
 - Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.
- Version 2.2.4
 - Fix MISRA-C 2012 issue.

enum _casper_operation
CASPER operation.

Values:

enumerator kCASPER_OpMul6464NoSum

enumerator kCASPER_OpMul6464Sum

Walking 1 or more of J loop, doing $r=a*b$ using $64 \times 64=128$

enumerator kCASPER_OpMul6464FullSum

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64 \times 64=128$, but assume inner j loop

enumerator kCASPER_OpMul6464Reduce

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64 \times 64=128$, but sum all of w.

enumerator kCASPER_OpAdd64

Walking 1 or more of J loop, doing $c,r[-1]=r+a*b$ using $64 \times 64=128$, but skip 1st write

enumerator kCASPER_OpSub64

Walking add with off_AB, and in/out off_RES doing $c,r=r+a+c$ using $64+64=65$

enumerator kCASPER_OpDouble64

Walking subtract with off_AB, and in/out off_RES doing $r=r-a$ using $64-64=64$, with last borrow implicit if any

enumerator kCASPER_OpXor64

Walking add to self with off_RES doing $c,r=r+r+c$ using $64+64=65$

enumerator kCASPER_OpRSub64

Walking XOR with off_AB, and in/out off_RES doing $r=r^a$ using $64^64=64$

enumerator kCASPER_OpShiftLeft32

Walking subtract with off_AB, and in/out off_RES using $r=a-r$

enumerator kCASPER_OpShiftRight32

Walking shift left doing $r1,r=(b*D)|r1$, where D is 2^{amt} and is loaded by app (off_CD not used)

enumerator kCASPER_OpCopy

Walking shift right doing $r, r1 = (b * D) | r1$, where D is $2^{(32 - \text{amt})}$ and is loaded by app (off_CD not used) and off_RES starts at MSW

enumerator kCASPER_OpRemask

Copy from ABoff to resoff, 64b at a time

enumerator kCASPER_OpFill

Copy and mask from ABoff to resoff, 64b at a time

enumerator kCASPER_OpZero

Fill RESOFF using 64 bits at a time with value in A and B

enumerator kCASPER_OpCompare

Fill RESOFF using 64 bits at a time of 0s

enumerator kCASPER_OpCompareFast

Compare two arrays, running all the way to the end

enum _casper_algo_t

Algorithm used for CASPER operation.

Values:

enumerator kCASPER_ECC_P256

ECC_P256

enumerator kCASPER_ECC_P384

ECC_P384

enumerator kCASPER_ECC_P521

ECC_P521

Values:

enumerator kCASPER_RamOffset_Result

enumerator kCASPER_RamOffset_Base

enumerator kCASPER_RamOffset_TempBase

enumerator kCASPER_RamOffset_Modulus

enumerator kCASPER_RamOffset_M64

typedef enum _casper_operation casper_operation_t

CASPER operation.

typedef enum _casper_algo_t casper_algo_t

Algorithm used for CASPER operation.

void CASPER_Init(CASPER_Type *base)

Enables clock and disables reset for CASPER peripheral.

Enable clock and disable reset for CASPER.

Parameters

- base – CASPER base address

void CASPER_Deinit(CASPER_Type *base)

Disables clock for CASPER peripheral.

Disable clock and enable reset.

Parameters

- base – CASPER base address

CASPER_CP

CASPER_CP_CTRL0

CASPER_CP_CTRL1

CASPER_CP_LOADER

CASPER_CP_STATUS

CASPER_CP_INTENSET

CASPER_CP_INTENCLR

CASPER_CP_INTSTAT

CASPER_CP_AREG

CASPER_CP_BREG

CASPER_CP_CREG

CASPER_CP_DREG

CASPER_CP_RES0

CASPER_CP_RES1

CASPER_CP_RES2

CASPER_CP_RES3

CASPER_CP_MASK

CASPER_CP_REMASK

CASPER_CP_LOCK

CASPER_CP_ID

CASPER_Wr32b(value, off)

CASPER_Wr64b(value, off)

CASPER_Rd32b(off)

N_wordlen_max

2.4 casper_driver_pkha

void CASPER_ModExp(CASPER_Type *base, const uint8_t *signature, const uint8_t *pubN, size_t wordLen, uint32_t pubE, uint8_t *plaintext)

Performs modular exponentiation - $(A^E) \bmod N$.

This function performs modular exponentiation.

Parameters

- base – CASPER base address

- signature – first addend (in little endian format)
- pubN – modulus (in little endian format)
- wordLen – Size of pubN in bytes
- pubE – exponent
- plaintext – **[out]** Output array to store result of operation (in little endian format)

void CASPER_ecc_init(*casper_algo_t* curve)

Initialize prime modulus mod in Casper memory .

Set the prime modulus mod in Casper memory and set N_wordlen according to selected algorithm.

Parameters

- curve – elliptic curve algorithm

void CASPER_ECC_SECP256R1_Mul(CASPER_Type *base, uint32_t resX[8], uint32_t resY[8],
uint32_t X[8], uint32_t Y[8], uint32_t scalar[8])

Performs ECC secp256r1 point single scalar multiplication.

This function performs ECC secp256r1 point single scalar multiplication [resX; resY] = scalar * [X; Y] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

void CASPER_ECC_SECP256R1_MulAdd(CASPER_Type *base, uint32_t resX[8], uint32_t
resY[8], uint32_t X1[8], uint32_t Y1[8], uint32_t
scalar1[8], uint32_t X2[8], uint32_t Y2[8], uint32_t
scalar2[8])

Performs ECC secp256r1 point double scalar multiplication.

This function performs ECC secp256r1 point double scalar multiplication [resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.

- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_SECP384R1_Mul(CASPER_Type *base, uint32_t resX[12], uint32_t resY[12],
                             uint32_t X[12], uint32_t Y[12], uint32_t scalar[12])
```

Performs ECC secp384r1 point single scalar multiplication.

This function performs ECC secp384r1 point single scalar multiplication $[resX; resY] = scalar * [X; Y]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

```
void CASPER_ECC_SECP384R1_MulAdd(CASPER_Type *base, uint32_t resX[12], uint32_t
                                resY[12], uint32_t X1[12], uint32_t Y1[12], uint32_t
                                scalar1[12], uint32_t X2[12], uint32_t Y2[12], uint32_t
                                scalar2[12])
```

Performs ECC secp384r1 point double scalar multiplication.

This function performs ECC secp384r1 point double scalar multiplication $[resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.
- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_SECP521R1_Mul(CASPER_Type *base, uint32_t resX[18], uint32_t resY[18],
                              uint32_t X[18], uint32_t Y[18], uint32_t scalar[18])
```

Performs ECC secp521r1 point single scalar multiplication.

This function performs ECC secp521r1 point single scalar multiplication $[resX; resY] = scalar * [X; Y]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address

- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

```
void CASPER_ECC_SECP521R1_MulAdd(CASPER_Type *base, uint32_t resX[18], uint32_t  
                                resY[18], uint32_t X1[18], uint32_t Y1[18], uint32_t  
                                scalar1[18], uint32_t X2[18], uint32_t Y2[18], uint32_t  
                                scalar2[18])
```

Performs ECC secp521r1 point double scalar multiplication.

This function performs ECC secp521r1 point double scalar multiplication [resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.
- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_equal(int *res, uint32_t *op1, uint32_t *op2)
```

```
void CASPER_ECC_equal_to_zero(int *res, uint32_t *op1)
```

2.5 CDOG

```
status_t CDOG_Init(CDOG_Type *base, cdog_config_t *conf)
```

Initialize CDOG.

This function initializes CDOG block and setting.

Parameters

- base – CDOG peripheral base address
- conf – CDOG configuration structure

Returns

Status of the init operation

```
void CDOG_Deinit(CDOG_Type *base)
```

Deinitialize CDOG.

This function deinitializes CDOG secure counter.

Parameters

- base – CDOG peripheral base address

void CDOG_GetDefaultConfig(*cdog_config_t* *conf)

Sets the default configuration of CDOG.

This function initialize CDOG config structure to default values.

Parameters

- conf – CDOG configuration structure

void CDOG_Stop(CDOG_Type *base, uint32_t stop)

Stops secure counter and instruction timer.

This function stops instruction timer and secure counter. This also change state of CDOG to IDLE.

Parameters

- base – CDOG peripheral base address
- stop – expected value which will be compared with value of secure counter

void CDOG_Start(CDOG_Type *base, uint32_t reload, uint32_t start)

Sets secure counter and instruction timer values.

This function sets value in RELOAD and START registers for instruction timer and secure counter

Parameters

- base – CDOG peripheral base address
- reload – reload value
- start – start value

void CDOG_Check(CDOG_Type *base, uint32_t check)

Checks secure counter.

This function compares stop value in handler with secure counter value by writing to RELOAD register.

Parameters

- base – CDOG peripheral base address
- check – expected (stop) value

void CDOG_Set(CDOG_Type *base, uint32_t stop, uint32_t reload, uint32_t start)

Sets secure counter and instruction timer values.

This function sets value in STOP, RELOAD and START registers for instruction timer and secure counter.

Parameters

- base – CDOG peripheral base address
- stop – expected value which will be compared with value of secure counter
- reload – reload value for instruction timer
- start – start value for secure timer

void CDOG_Add(CDOG_Type *base, uint32_t add)

Add value to secure counter.

This function add specified value to secure counter.

Parameters

- base – CDOG peripheral base address.
- add – Value to be added.

void CDOG_Add1(CDOG_Type *base)

Add 1 to secure counter.

This function add 1 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_Add16(CDOG_Type *base)

Add 16 to secure counter.

This function add 16 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_Add256(CDOG_Type *base)

Add 256 to secure counter.

This function add 256 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_Sub(CDOG_Type *base, uint32_t sub)

brief Subtract value to secure counter

This function subtract specified value to secure counter.

param base CDOG peripheral base address. param sub Value to be subtracted.

void CDOG_Sub1(CDOG_Type *base)

Subtract 1 from secure counter.

This function subtract specified 1 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_Sub16(CDOG_Type *base)

Subtract 16 from secure counter.

This function subtract specified 16 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_Sub256(CDOG_Type *base)

Subtract 256 from secure counter.

This function subtract specified 256 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG_WritePersistent(CDOG_Type *base, uint32_t value)

Set the CDOG persistent word.

Parameters

- base – CDOG peripheral base address.

- value – The value to be written.

uint32_t CDOG_ReadPersistent(CDOG_Type *base)

Get the CDOG persistent word.

Parameters

- base – CDOG peripheral base address.

Returns

The persistent word.

FSL_CDOG_DRIVER_VERSION

Defines CDOG driver version 2.1.3.

Change log:

- Version 2.1.3
 - Re-design multiple instance IRQs and Clocks
 - Add fix for RESTART command errata
- Version 2.1.2
 - Support multiple IRQs
 - Fix default CONTROL values
- Version 2.1.1
 - Remove bit CONTROL[CONTROL_CTRL]
- Version 2.1.0
 - Rename CWT to CDOG
- Version 2.0.2
 - Fix MISRA-2012 issues
- Version 2.0.1
 - Fix doxygen issues
- Version 2.0.0
 - initial version

enum __cdog_debug_Action_ctrl_enum

Values:

enumerator kCDOG_DebugHaltCtrl_Run

enumerator kCDOG_DebugHaltCtrl_Pause

enum __cdog_irq_pause_ctrl_enum

Values:

enumerator kCDOG_IrqPauseCtrl_Run

enumerator kCDOG_IrqPauseCtrl_Pause

enum __cdog_fault_ctrl_enum

Values:

enumerator kCDOG_FaultCtrl_EnableReset

enumerator kCDOG_FaultCtrl_EnableInterrupt

```
    enumerator kCDOG_FaultCtrl_NoAction

enum __code_lock_ctrl_enum
    Values:
        enumerator kCDOG_LockCtrl_Lock
        enumerator kCDOG_LockCtrl_Unlock

typedef uint32_t secure_counter_t

SC_ADD(add)
SC_ADD1
SC_ADD16
SC_ADD256
SC_SUB(sub)
SC_SUB1
SC_SUB16
SC_SUB256
SC_CHECK(val)

struct cdog_config_t
    #include <fsl_cdog.h>
```

2.6 CRC: Cyclic Redundancy Check Driver

```
FSL_CRC_DRIVER_VERSION
CRC driver version. Version 2.1.1.
Current version: 2.1.1
Change log:
    • Version 2.0.0
        – initial version
    • Version 2.0.1
        – add explicit type cast when writing to WR_DATA
    • Version 2.0.2
        – Fix MISRA issue
    • Version 2.1.0
        – Add CRC_WriteSeed function
    • Version 2.1.1
        – Fix MISRA issue

enum __crc_polynomial
    CRC polynomials to use.
    Values:
```



```
enumerator kCRC_Polynomial_CRC_CCITT
    x^16+x^12+x^5+1
```

```
enumerator kCRC_Polynomial_CRC_16
    x^16+x^15+x^2+1
```

```
enumerator kCRC_Polynomial_CRC_32
    x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1
```

```
typedef enum _crc_polynomial crc_polynomial_t
    CRC polynomials to use.
```

```
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 functions enables the CRC peripheral clock in the LPC SYSCON block. It also configures the CRC engine and starts 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 functions disables the CRC peripheral clock in the LPC SYSCON block.

Parameters

- base – CRC peripheral address.

```
void CRC_Reset(CRC_Type *base)
    resets CRC peripheral module.
```

Parameters

- base – CRC peripheral address.

```
void CRC_WriteSeed(CRC_Type *base, uint32_t seed)
    Write seed to CRC peripheral module.
```

Parameters

- base – CRC peripheral address.
- seed – CRC Seed value.

```
void CRC_GetDefaultConfig(crc_config_t *config)
    Loads default values to CRC protocol configuration structure.
```

Loads default values to CRC protocol configuration structure. The default values are:

```
config->polynomial = kCRC_Polynomial_CRC_CCITT;
config->reverseIn = false;
config->complementIn = false;
config->reverseOut = false;
config->complementOut = false;
config->seed = 0xFFFFU;
```

Parameters

- `config` – CRC protocol configuration structure

`void CRC_GetConfig(CRC_Type *base, crc_config_t *config)`

Loads actual values configured in CRC peripheral to CRC protocol configuration structure.

The values, including seed, can be used to resume CRC calculation later.

Parameters

- `base` – CRC peripheral address.
- `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 CRC data register.

Parameters

- `base` – CRC peripheral address.
- `data` – Input data stream, MSByte in `data[0]`.
- `dataSize` – Size of the input data buffer in bytes.

`static inline uint32_t CRC_Get32bitResult(CRC_Type *base)`

Reads 32-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

final 32-bit checksum, after configured bit reverse and complement operations.

`static inline uint16_t CRC_Get16bitResult(CRC_Type *base)`

Reads 16-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

final 16-bit checksum, after configured bit reverse and complement operations.

`CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT`

Default configuration structure filled by `CRC_GetDefaultConfig()`. Uses CRC-16/CCITT-FALSE as default.

`struct _crc_config`

`#include <fsl_crc.h>` CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

Public Members

`crc_polynomial_t` polynomial

CRC polynomial.

`bool` reverseIn

Reverse bits on input.

`bool complementIn`
Perform 1's complement on input.

`bool reverseOut`
Reverse bits on output.

`bool complementOut`
Perform 1's complement on output.

`uint32_t seed`
Starting checksum value.

2.7 CTIMER: Standard counter/timers

`void CTIMER_Init(CTIMER_Type *base, const ctimer_config_t *config)`
Ungates the clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application before using the driver.

Parameters

- `base` – Ctimer peripheral base address
- `config` – Pointer to the user configuration structure.

`void CTIMER_Deinit(CTIMER_Type *base)`
Gates the timer clock.

Parameters

- `base` – Ctimer peripheral base address

`void CTIMER_GetDefaultConfig(ctimer_config_t *config)`
Fills in the timers configuration structure with the default settings.

The default values are:

```
config->mode = kCTIMER_TimerMode;
config->input = kCTIMER_Capture_0;
config->prescale = 0;
```

Parameters

- `config` – Pointer to the user configuration structure.

`status_t CTIMER_SetupPwmPeriod(CTIMER_Type *base, const ctimer_match_t pwmPeriodChannel, ctimer_match_t matchChannel, uint32_t pwmPeriod, uint32_t pulsePeriod, bool enableInt)`

Configures the PWM signal parameters.

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function can manually assign the specified channel to set the PWM cycle.

Note: When setting PWM output from multiple output pins, all should use the same PWM period

Parameters

- base – Ctimer peripheral base address
- pwmPeriodChannel – Specify the channel to control the PWM period
- matchChannel – Match pin to be used to output the PWM signal
- pwmPeriod – PWM period match value
- pulsePeriod – Pulse width match value
- enableInt – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

Returns

kStatus_Success on success kStatus_Fail If matchChannel is equal to pwmPeriodChannel; this channel is reserved to set the PWM cycle If PWM pulse width register value is larger than 0xFFFFFFFF.

```
status_t CTIMER_SetupPwm(CTIMER_Type *base, const ctimer_match_t pwmPeriodChannel,  
                        ctimer_match_t matchChannel, uint8_t dutyCyclePercent, uint32_t  
                        pwmFreq_Hz, uint32_t srcClock_Hz, bool enableInt)
```

Configures the PWM signal parameters.

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function can manually assign the specified channel to set the PWM cycle.

Note: When setting PWM output from multiple output pins, all should use the same PWM frequency. Please use CTIMER_SetupPwmPeriod to set up the PWM with high resolution.

Parameters

- base – Ctimer peripheral base address
- pwmPeriodChannel – Specify the channel to control the PWM period
- matchChannel – Match pin to be used to output the PWM signal
- dutyCyclePercent – PWM pulse width; the value should be between 0 to 100
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – Timer counter clock in Hz
- enableInt – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

```
static inline void CTIMER_UpdatePwmPulsePeriod(CTIMER_Type *base, ctimer_match_t  
                                              matchChannel, uint32_t pulsePeriod)
```

Updates the pulse period of an active PWM signal.

Parameters

- base – Ctimer peripheral base address
- matchChannel – Match pin to be used to output the PWM signal
- pulsePeriod – New PWM pulse width match value

```
status_t CTIMER_UpdatePwmDutycycle(CTIMER_Type *base, const ctimer_match_t  
                                  pwmPeriodChannel, ctimer_match_t matchChannel,  
                                  uint8_t dutyCyclePercent)
```

Updates the duty cycle of an active PWM signal.

Note: Please use `CTIMER_SetupPwmPeriod` to update the PWM with high resolution. This function can manually assign the specified channel to set the PWM cycle.

Parameters

- `base` – Ctimer peripheral base address
- `pwmPeriodChannel` – Specify the channel to control the PWM period
- `matchChannel` – Match pin to be used to output the PWM signal
- `dutyCyclePercent` – New PWM pulse width; the value should be between 0 to 100

Returns

`kStatus_Success` on success `kStatus_Fail` If PWM pulse width register value is larger than `0xFFFFFFFF`.

```
static inline void CTIMER_EnableInterrupts(CTIMER_Type *base, uint32_t mask)
```

Enables the selected Timer interrupts.

Parameters

- `base` – Ctimer peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `ctimer_interrupt_enable_t`

```
static inline void CTIMER_DisableInterrupts(CTIMER_Type *base, uint32_t mask)
```

Disables the selected Timer interrupts.

Parameters

- `base` – Ctimer peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `ctimer_interrupt_enable_t`

```
static inline uint32_t CTIMER_GetEnabledInterrupts(CTIMER_Type *base)
```

Gets the enabled Timer interrupts.

Parameters

- `base` – Ctimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `ctimer_interrupt_enable_t`

```
static inline uint32_t CTIMER_GetStatusFlags(CTIMER_Type *base)
```

Gets the Timer status flags.

Parameters

- `base` – Ctimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `ctimer_status_flags_t`

```
static inline void CTIMER_ClearStatusFlags(CTIMER_Type *base, uint32_t mask)
```

Clears the Timer status flags.

Parameters

- `base` – Ctimer peripheral base address

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

static inline void CTIMER_StartTimer(CTIMER_Type *base)

Starts the Timer counter.

Parameters

- base – Ctimer peripheral base address

static inline void CTIMER_StopTimer(CTIMER_Type *base)

Stops the Timer counter.

Parameters

- base – Ctimer peripheral base address

FSL_CTIMER_DRIVER_VERSION

Version 2.3.3

enum _ctimer_capture_channel

List of Timer capture channels.

Values:

enumerator kCTIMER_Capture_0

Timer capture channel 0

enumerator kCTIMER_Capture_1

Timer capture channel 1

enumerator kCTIMER_Capture_3

Timer capture channel 3

enum _ctimer_capture_edge

List of capture edge options.

Values:

enumerator kCTIMER_Capture_RiseEdge

Capture on rising edge

enumerator kCTIMER_Capture_FallEdge

Capture on falling edge

enumerator kCTIMER_Capture_BothEdge

Capture on rising and falling edge

enum _ctimer_match

List of Timer match registers.

Values:

enumerator kCTIMER_Match_0

Timer match register 0

enumerator kCTIMER_Match_1

Timer match register 1

enumerator kCTIMER_Match_2

Timer match register 2

enumerator kCTIMER_Match_3

Timer match register 3

enum _ctimer_external_match

List of external match.

Values:

enumerator kCTIMER_External_Match_0
External match 0

enumerator kCTIMER_External_Match_1
External match 1

enumerator kCTIMER_External_Match_2
External match 2

enumerator kCTIMER_External_Match_3
External match 3

enum _ctimer_match_output_control

List of output control options.

Values:

enumerator kCTIMER_Output_NoAction
No action is taken

enumerator kCTIMER_Output_Clear
Clear the EM bit/output to 0

enumerator kCTIMER_Output_Set
Set the EM bit/output to 1

enumerator kCTIMER_Output_Toggle
Toggle the EM bit/output

enum _ctimer_timer_mode

List of Timer modes.

Values:

enumerator kCTIMER_TimerMode

enumerator kCTIMER_IncreaseOnRiseEdge

enumerator kCTIMER_IncreaseOnFallEdge

enumerator kCTIMER_IncreaseOnBothEdge

enum _ctimer_interrupt_enable

List of Timer interrupts.

Values:

enumerator kCTIMER_Match0InterruptEnable
Match 0 interrupt

enumerator kCTIMER_Match1InterruptEnable
Match 1 interrupt

enumerator kCTIMER_Match2InterruptEnable
Match 2 interrupt

enumerator kCTIMER_Match3InterruptEnable
Match 3 interrupt

enum `_ctimer_status_flags`

List of Timer flags.

Values:

enumerator `kCTIMER_Match0Flag`

Match 0 interrupt flag

enumerator `kCTIMER_Match1Flag`

Match 1 interrupt flag

enumerator `kCTIMER_Match2Flag`

Match 2 interrupt flag

enumerator `kCTIMER_Match3Flag`

Match 3 interrupt flag

enum `ctimer_callback_type_t`

Callback type when registering for a callback. When registering a callback an array of function pointers is passed the size could be 1 or 8, the callback type will tell that.

Values:

enumerator `kCTIMER_SingleCallback`

Single Callback type where there is only one callback for the timer. based on the status flags different channels needs to be handled differently

enumerator `kCTIMER_MultipleCallback`

Multiple Callback type where there can be 8 valid callbacks, one per channel. for both match/capture

typedef enum `_ctimer_capture_channel` `ctimer_capture_channel_t`

List of Timer capture channels.

typedef enum `_ctimer_capture_edge` `ctimer_capture_edge_t`

List of capture edge options.

typedef enum `_ctimer_match` `ctimer_match_t`

List of Timer match registers.

typedef enum `_ctimer_external_match` `ctimer_external_match_t`

List of external match.

typedef enum `_ctimer_match_output_control` `ctimer_match_output_control_t`

List of output control options.

typedef enum `_ctimer_timer_mode` `ctimer_timer_mode_t`

List of Timer modes.

typedef enum `_ctimer_interrupt_enable` `ctimer_interrupt_enable_t`

List of Timer interrupts.

typedef enum `_ctimer_status_flags` `ctimer_status_flags_t`

List of Timer flags.

typedef void (`*ctimer_callback_t`)(`uint32_t` flags)

typedef struct `_ctimer_match_config` `ctimer_match_config_t`

Match configuration.

This structure holds the configuration settings for each match register.


```
typedef struct _ctimer_config ctimer_config_t
```

Timer configuration structure.

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the CTIMER_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

```
void CTIMER_SetupMatch(CTIMER_Type *base, ctimer_match_t matchChannel, const  
                      ctimer_match_config_t *config)
```

Setup the match register.

User configuration is used to setup the match value and action to be taken when a match occurs.

Parameters

- base – Ctimer peripheral base address
- matchChannel – Match register to configure
- config – Pointer to the match configuration structure

```
uint32_t CTIMER_GetOutputMatchStatus(CTIMER_Type *base, uint32_t matchChannel)
```

Get the status of output match.

This function gets the status of output MAT, whether or not this output is connected to a pin. This status is driven to the MAT pins if the match function is selected via IOCON. 0 = LOW. 1 = HIGH.

Parameters

- base – Ctimer peripheral base address
- matchChannel – External match channel, user can obtain the status of multiple match channels at the same time by using the logic of “|” enumeration *ctimer_external_match_t*

Returns

The mask of external match channel status flags. Users need to use the *_ctimer_external_match_t* type to decode the return variables.

```
void CTIMER_SetupCapture(CTIMER_Type *base, ctimer_capture_channel_t capture,  
                        ctimer_capture_edge_t edge, bool enableInt)
```

Setup the capture.

Parameters

- base – Ctimer peripheral base address
- capture – Capture channel to configure
- edge – Edge on the channel that will trigger a capture
- enableInt – Flag to enable channel interrupts, if enabled then the registered call back is called upon capture

```
static inline uint32_t CTIMER_GetTimerCountValue(CTIMER_Type *base)
```

Get the timer count value from TC register.

Parameters

- base – Ctimer peripheral base address.

Returns

return the timer count value.

```
void CTIMER_RegisterCallBack(CTIMER_Type *base, ctimer_callback_t *cb_func,  
                             ctimer_callback_type_t cb_type)
```

Register callback.

This function configures CTimer Callback in following modes:

- Single Callback: *cb_func* should be pointer to callback function pointer
For example: *ctimer_callback_t* *ctimer_callback* = *pwm_match_callback*;
CTIMER_RegisterCallBack(*CTIMER*, &*ctimer_callback*, *kCTIMER_SingleCallback*);
- Multiple Callback: *cb_func* should be pointer to array of callback function pointers Each element corresponds to Interrupt Flag in IR register.
For example: *ctimer_callback_t* *ctimer_callback_table*[] = {
ctimer_match0_callback, NULL, NULL, *ctimer_match3_callback*, NULL, NULL,
NULL, NULL}; *CTIMER_RegisterCallBack*(*CTIMER*, &*ctimer_callback_table*[0], *kCTIMER_MultipleCallback*);

Parameters

- *base* – Ctimer peripheral base address
- *cb_func* – Pointer to callback function pointer
- *cb_type* – callback function type, singular or multiple

```
static inline void CTIMER_Reset(CTIMER_Type *base)
```

Reset the counter.

The timer counter and prescale counter are reset on the next positive edge of the APB clock.

Parameters

- *base* – Ctimer peripheral base address

```
static inline void CTIMER_SetPrescale(CTIMER_Type *base, uint32_t prescale)
```

Setup the timer prescale value.

Specifies the maximum value for the Prescale Counter.

Parameters

- *base* – Ctimer peripheral base address
- *prescale* – Prescale value

```
static inline uint32_t CTIMER_GetCaptureValue(CTIMER_Type *base, ctimer_capture_channel_t  
                                              capture)
```

Get capture channel value.

Get the counter/timer value on the corresponding capture channel.

Parameters

- *base* – Ctimer peripheral base address
- *capture* – Select capture channel

Returns

The timer count capture value.

```
static inline void CTIMER_EnableResetMatchChannel(CTIMER_Type *base, ctimer_match_t  
                                                  match, bool enable)
```

Enable reset match channel.

Set the specified match channel reset operation.

Parameters

- *base* – Ctimer peripheral base address

- `match` – match channel used
- `enable` – Enable match channel reset operation.

```
static inline void CTIMER_EnableStopMatchChannel(CTIMER_Type *base, ctimer_match_t
                                                match, bool enable)
```

Enable stop match channel.

Set the specified match channel stop operation.

Parameters

- `base` – Ctimer peripheral base address.
- `match` – match channel used.
- `enable` – Enable match channel stop operation.

```
static inline void CTIMER_EnableMatchChannelReload(CTIMER_Type *base, ctimer_match_t
                                                  match, bool enable)
```

Enable reload channel falling edge.

Enable the specified match channel reload match shadow value.

Parameters

- `base` – Ctimer peripheral base address.
- `match` – match channel used.
- `enable` – Enable .

```
static inline void CTIMER_EnableRisingEdgeCapture(CTIMER_Type *base,
                                                  ctimer_capture_channel_t capture, bool
                                                  enable)
```

Enable capture channel rising edge.

Sets the specified capture channel for rising edge capture.

Parameters

- `base` – Ctimer peripheral base address.
- `capture` – capture channel used.
- `enable` – Enable rising edge capture.

```
static inline void CTIMER_EnableFallingEdgeCapture(CTIMER_Type *base,
                                                  ctimer_capture_channel_t capture, bool
                                                  enable)
```

Enable capture channel falling edge.

Sets the specified capture channel for falling edge capture.

Parameters

- `base` – Ctimer peripheral base address.
- `capture` – capture channel used.
- `enable` – Enable falling edge capture.

```
static inline void CTIMER_SetShadowValue(CTIMER_Type *base, ctimer_match_t match,
                                         uint32_t matchvalue)
```

Set the specified match shadow channel.

Parameters

- `base` – Ctimer peripheral base address.
- `match` – match channel used.

- `matchvalue` – Reload the value of the corresponding match register.

`struct __ctimer_match_config`

#include <fsl_ctimer.h> Match configuration.

This structure holds the configuration settings for each match register.

Public Members

`uint32_t matchValue`

This is stored in the match register

`bool enableCounterReset`

true: Match will reset the counter false: Match will not reset the counter

`bool enableCounterStop`

true: Match will stop the counter false: Match will not stop the counter

`ctimer_match_output_control_t outControl`

Action to be taken on a match on the EM bit/output

`bool outPinInitState`

Initial value of the EM bit/output

`bool enableInterrupt`

true: Generate interrupt upon match false: Do not generate interrupt on match

`struct __ctimer_config`

#include <fsl_ctimer.h> Timer configuration structure.

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the `CTIMER_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

`ctimer_timer_mode_t mode`

Timer mode

`ctimer_capture_channel_t input`

Input channel to increment the timer; used only in timer modes that rely on this input signal to increment TC

`uint32_t prescale`

Prescale value

2.8 DMA: Direct Memory Access Controller Driver

`void DMA_Init(DMA_Type *base)`

Initializes DMA peripheral.

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

- `base` – DMA peripheral base address.

`void DMA_Deinit(DMA_Type *base)`

Deinitializes DMA peripheral.

This function gates the DMA clock.

Parameters

- `base` – DMA peripheral base address.

`void DMA_InstallDescriptorMemory(DMA_Type *base, void *addr)`

Install DMA descriptor memory.

This function used to register DMA descriptor memory for linked transfer, a typical case is ping pong transfer which will request more than one DMA descriptor memory space, although current DMA driver has a default DMA descriptor buffer, but it support one DMA descriptor for one channel only.

Parameters

- `base` – DMA base address.
- `addr` – DMA descriptor address

`static inline bool DMA_ChannelIsActive(DMA_Type *base, uint32_t channel)`

Return whether DMA channel is processing transfer.

Parameters

- `base` – DMA peripheral base address.
- `channel` – DMA channel number.

Returns

True for active state, false otherwise.

`static inline bool DMA_ChannelIsBusy(DMA_Type *base, uint32_t channel)`

Return whether DMA channel is busy.

Parameters

- `base` – DMA peripheral base address.
- `channel` – DMA channel number.

Returns

True for busy state, false otherwise.

`static inline void DMA_EnableChannelInterrupts(DMA_Type *base, uint32_t channel)`

Enables the interrupt source for the DMA transfer.

Parameters

- `base` – DMA peripheral base address.
- `channel` – DMA channel number.

`static inline void DMA_DisableChannelInterrupts(DMA_Type *base, uint32_t channel)`

Disables the interrupt source for the DMA transfer.

Parameters

- `base` – DMA peripheral base address.
- `channel` – DMA channel number.

`static inline void DMA_EnableChannel(DMA_Type *base, uint32_t channel)`

Enable DMA channel.

Parameters

- `base` – DMA peripheral base address.

- channel – DMA channel number.

static inline void DMA_DisableChannel(DMA_Type *base, uint32_t channel)
Disable DMA channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_EnableChannelPeriphRq(DMA_Type *base, uint32_t channel)
Set PERIPHREQEN of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannelPeriphRq(DMA_Type *base, uint32_t channel)
Get PERIPHREQEN value of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

True for enabled PeriphRq, false for disabled.

void DMA_ConfigureChannelTrigger(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger)
Set trigger settings of DMA channel.

Deprecated:

Do not use this function. It has been superceded by DMA_SetChannelConfig.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- trigger – trigger configuration.

void DMA_SetChannelConfig(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger, bool isPeriph)

set channel config.

This function provide a interface to configure channel configuration registers.

Parameters

- base – DMA base address.
- channel – DMA channel number.
- trigger – channel configurations structure.
- isPeriph – true is periph request, false is not.

static inline uint32_t DMA_SetChannelXferConfig(bool reload, bool clrTrig, bool intA, bool intB, uint8_t width, uint8_t srcInc, uint8_t dstInc, uint32_t bytes)

DMA channel xfer transfer configurations.

Parameters

- reload – true is reload link descriptor after current exhaust, false is not
- clrTrig – true is clear trigger status, wait software trigger, false is not
- intA – enable interruptA
- intB – enable interruptB
- width – transfer width
- srcInc – source address interleave size
- dstInc – destination address interleave size
- bytes – transfer bytes

Returns

The vaule of xfer config

uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)

Gets the remaining bytes of the current DMA descriptor transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

The number of bytes which have not been transferred yet.

static inline void DMA_SetChannelPriority(DMA_Type *base, uint32_t channel, dma_priority_t priority)

Set priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- priority – Channel priority value.

static inline dma_priority_t DMA_GetChannelPriority(DMA_Type *base, uint32_t channel)

Get priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

Channel priority value.

static inline void DMA_SetChannelConfigValid(DMA_Type *base, uint32_t channel)

Set channel configuration valid.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DoChannelSoftwareTrigger(DMA_Type *base, uint32_t channel)

Do software trigger for the channel.

Parameters

- base – DMA peripheral base address.

- channel – DMA channel number.

```
static inline void DMA_LoadChannelTransferConfig(DMA_Type *base, uint32_t channel, uint32_t xfer)
```

Load channel transfer configurations.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- xfer – transfer configurations.

```
void DMA_CreateDescriptor(dma_descriptor_t *desc, dma_xfercfg_t *xfercfg, void *srcAddr, void *dstAddr, void *nextDesc)
```

Create application specific DMA descriptor to be used in a chain in transfer.

Deprecated:

Do not use this function. It has been superseded by DMA_SetupDescriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcAddr – Address of last item to transmit
- dstAddr – Address of last item to receive.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr, void *dstStartAddr, void *nextDesc)
```

setup dma descriptor

Note: This function do not support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupChannelDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr, void *dstStartAddr, void *nextDesc, dma_burst_wrap_t wrapType, uint32_t burstSize)
```

setup dma channel descriptor

Note: This function support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.

- wrapType – burst wrap type.
- burstSize – burst size, reference `_dma_burst_size`.

`void DMA_LoadChannelDescriptor(DMA_Type *base, uint32_t channel, dma_descriptor_t *descriptor)`

load channel transfer decriptor.

This function can be used to load desscriptor to driver internal channel descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- for the polling transfer, application can allocate a local descriptor memory table to prepare a descriptor firstly and then call this api to load the configured descriptor to driver descriptor table.

```
DMA_Init(DMA0);
DMA_EnableChannel(DMA0, DEMO_DMA_CHANNEL);
DMA_SetupDescriptor(desc, xferCfg, s_srcBuffer, &s_destBuffer[0], NULL);
DMA_LoadChannelDescriptor(DMA0, DEMO_DMA_CHANNEL, (dma_descriptor_t *)desc);
DMA_DoChannelSoftwareTrigger(DMA0, DEMO_DMA_CHANNEL);
while(DMA_ChannelIsBusy(DMA0, DEMO_DMA_CHANNEL))
{ }
```

Parameters

- base – DMA base address.
- channel – DMA channel.
- descriptor – configured DMA descriptor.

`void DMA_AbortTransfer(dma_handle_t *handle)`

Abort running transfer by handle.

This function aborts DMA transfer specified by handle.

Parameters

- handle – DMA handle pointer.

`void DMA_CreateHandle(dma_handle_t *handle, DMA_Type *base, uint32_t channel)`

Creates the DMA handle.

This function is called if using transaction API for DMA. This function initializes the internal state of DMA handle.

Parameters

- handle – DMA handle pointer. The DMA handle stores callback function and parameters.
- base – DMA peripheral base address.
- channel – DMA channel number.

`void DMA_SetCallback(dma_handle_t *handle, dma_callback callback, void *userData)`

Installs a callback function for the DMA transfer.

This callback is called in DMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

- handle – DMA handle pointer.
- callback – DMA callback function pointer.
- userData – Parameter for callback function.

```
void DMA__PrepareTransfer(dma_transfer_config_t *config, void *srcAddr, void *dstAddr,  
                        uint32_t byteWidth, uint32_t transferBytes, dma_transfer_type_t  
                        type, void *nextDesc)
```

Prepares the DMA transfer structure.

Deprecated:

Do not use this function. It has been superceded by DMA_PrepareChannelTransfer. 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, so the source address must be 4 bytes aligned, or it shall result in source address error(SAE).

Parameters

- config – The user configuration structure of type *dma_transfer_t*.
- srcAddr – DMA transfer source address.
- dstAddr – DMA transfer destination address.
- byteWidth – DMA transfer destination address width(bytes).
- transferBytes – DMA transfer bytes to be transferred.
- type – DMA transfer type.
- nextDesc – Chain custom descriptor to transfer.

```
void DMA__PrepareChannelTransfer(dma_channel_config_t *config, void *srcStartAddr, void  
                               *dstStartAddr, uint32_t xferCfg, dma_transfer_type_t type,  
                               dma_channel_trigger_t *trigger, void *nextDesc)
```

Prepare channel transfer configurations.

This function used to prepare channel transfer configurations.

Parameters

- config – Pointer to DMA channel transfer configuration structure.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- type – transfer type.
- trigger – DMA channel trigger configurations.
- nextDesc – address of next descriptor.

```
status_t DMA__SubmitTransfer(dma_handle_t *handle, dma_transfer_config_t *config)
```

Submits the DMA transfer request.

Deprecated:

Do not use this function. It has been superceded by DMA_SubmitChannelTransfer.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an un-processed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- kStatus_DMA_Success – It means submit transfer request succeed.
- kStatus_DMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_DMA_Busy – It means the given channel is busy, need to submit request later.

```
void DMA_SubmitChannelTransferParameter(dma_handle_t *handle, uint32_t xferCfg, void
                                     *srcStartAddr, void *dstStartAddr, void *nextDesc)
```

Submit channel transfer paramter directly.

This function used to configue channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,
↪intA, intB, width, srcInc, dstInc,
↪bytes), srcStartAddr, dstStartAddr, NULL);
DMA_StartTransfer(handle)
```

- for the linked transfer, application should responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[3]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,
↪intA, intB, width, srcInc, dstInc,
↪bytes), srcStartAddr, dstStartAddr, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.

- nextDesc – address of next descriptor.

void DMA_SubmitChannelDescriptor(*dma_handle_t* *handle, *dma_descriptor_t* *descriptor)

Submit channel descriptor.

This function used to configure channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, this function is typical for the ping pong case:

- for the ping pong case, application should responsible for the descriptor, for example, application should prepare two descriptor table with macro.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[2]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelDescriptor(handle, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- descriptor – descriptor to submit.

status_t DMA_SubmitChannelTransfer(*dma_handle_t* *handle, *dma_channel_config_t* *config)

Submits the DMA channel transfer request.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits 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. It is used for the case:

- for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,NULL);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

- for the linked transfer, application should responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);
DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
```

(continues on next page)

(continued from previous page)

```
srcStartAddr, dstStartAddr, NULL);
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪nextDesc0);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

- c. for the ping pong case, application should responsible for link descriptor, for example, application should prepare two descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪nextDesc0);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- kStatus_DMA_Success – It means submit transfer request succeed.
- kStatus_DMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_DMA_Busy – It means the given channel is busy, need to submit request later.

void DMA_StartTransfer(*dma_handle_t* *handle)

DMA start transfer.

This function enables the channel request. User can call this function after submitting the transfer request It will trigger transfer start with software trigger only when hardware trigger is not used.

Parameters

- handle – DMA handle pointer.

void DMA_IRQHandle(DMA_Type *base)

DMA IRQ handler for descriptor transfer complete.

This function clears the channel major interrupt flag and call the callback function if it is not NULL.

Parameters

- base – DMA base address.

FSL_DMA_DRIVER_VERSION

DMA driver version.

Version 2.5.3.

_dma_transfer_status DMA transfer status

Values:

enumerator kStatus_DMA_Busy

Channel is busy and can't handle the transfer request.

_dma_addr_interleave_size dma address interleave size

Values:

enumerator kDMA_AddressInterleave0xWidth

dma source/destination address no interleave

enumerator kDMA_AddressInterleave1xWidth

dma source/destination address interleave 1xwidth

enumerator kDMA_AddressInterleave2xWidth

dma source/destination address interleave 2xwidth

enumerator kDMA_AddressInterleave4xWidth

dma source/destination address interleave 3xwidth

_dma_transfer_width dma transfer width

Values:

enumerator kDMA_Transfer8BitWidth

dma channel transfer bit width is 8 bit

enumerator kDMA_Transfer16BitWidth

dma channel transfer bit width is 16 bit

enumerator kDMA_Transfer32BitWidth

dma channel transfer bit width is 32 bit

enum _dma_priority

DMA channel priority.

Values:

enumerator kDMA_ChannelPriority0

Highest channel priority - priority 0

enumerator kDMA_ChannelPriority1

Channel priority 1

enumerator kDMA_ChannelPriority2

Channel priority 2

enumerator kDMA_ChannelPriority3

Channel priority 3

enumerator kDMA_ChannelPriority4

Channel priority 4

enumerator kDMA_ChannelPriority5

Channel priority 5

enumerator kDMA_ChannelPriority6

Channel priority 6

enumerator kDMA_ChannelPriority7

Lowest channel priority - priority 7

enum _dma_int

DMA interrupt flags.

Values:

enumerator kDMA_IntA

DMA interrupt flag A

enumerator kDMA_IntB

DMA interrupt flag B

enumerator kDMA_IntError

DMA interrupt flag error

enum _dma_trigger_type

DMA trigger type.

Values:

enumerator kDMA_NoTrigger

Trigger is disabled

enumerator kDMA_LowLevelTrigger

Low level active trigger

enumerator kDMA_HighLevelTrigger

High level active trigger

enumerator kDMA_FallingEdgeTrigger

Falling edge active trigger

enumerator kDMA_RisingEdgeTrigger

Rising edge active trigger

_dma_burst_size DMA burst size

Values:

enumerator kDMA_BurstSize1

burst size 1 transfer

enumerator kDMA_BurstSize2

burst size 2 transfer

enumerator kDMA_BurstSize4

burst size 4 transfer

enumerator kDMA_BurstSize8

burst size 8 transfer

enumerator kDMA_BurstSize16

burst size 16 transfer

enumerator kDMA__BurstSize32

burst size 32 transfer

enumerator kDMA__BurstSize64

burst size 64 transfer

enumerator kDMA__BurstSize128

burst size 128 transfer

enumerator kDMA__BurstSize256

burst size 256 transfer

enumerator kDMA__BurstSize512

burst size 512 transfer

enumerator kDMA__BurstSize1024

burst size 1024 transfer

enum __dma__trigger__burst

DMA trigger burst.

Values:

enumerator kDMA__SingleTransfer

Single transfer

enumerator kDMA__LevelBurstTransfer

Burst transfer driven by level trigger

enumerator kDMA__EdgeBurstTransfer1

Perform 1 transfer by edge trigger

enumerator kDMA__EdgeBurstTransfer2

Perform 2 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer4

Perform 4 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer8

Perform 8 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer16

Perform 16 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer32

Perform 32 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer64

Perform 64 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer128

Perform 128 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer256

Perform 256 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer512

Perform 512 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer1024

Perform 1024 transfers by edge trigger

enum *_dma_burst_wrap*

DMA burst wrapping.

Values:

enumerator *kDMA_NoWrap*

Wrapping is disabled

enumerator *kDMA_SrcWrap*

Wrapping is enabled for source

enumerator *kDMA_DstWrap*

Wrapping is enabled for destination

enumerator *kDMA_SrcAndDstWrap*

Wrapping is enabled for source and destination

enum *_dma_transfer_type*

DMA transfer type.

Values:

enumerator *kDMA_MemoryToMemory*

Transfer from memory to memory (increment source and destination)

enumerator *kDMA_PeripheralToMemory*

Transfer from peripheral to memory (increment only destination)

enumerator *kDMA_MemoryToPeripheral*

Transfer from memory to peripheral (increment only source)

enumerator *kDMA_StaticToStatic*

Peripheral to static memory (do not increment source or destination)

typedef struct *_dma_descriptor* *dma_descriptor_t*

DMA descriptor structure.

typedef struct *_dma_xfercfg* *dma_xfercfg_t*

DMA transfer configuration.

typedef enum *_dma_priority* *dma_priority_t*

DMA channel priority.

typedef enum *_dma_int* *dma_irq_t*

DMA interrupt flags.

typedef enum *_dma_trigger_type* *dma_trigger_type_t*

DMA trigger type.

typedef enum *_dma_trigger_burst* *dma_trigger_burst_t*

DMA trigger burst.

typedef enum *_dma_burst_wrap* *dma_burst_wrap_t*

DMA burst wrapping.

typedef enum *_dma_transfer_type* *dma_transfer_type_t*

DMA transfer type.

typedef struct *_dma_channel_trigger* *dma_channel_trigger_t*

DMA channel trigger.

typedef struct *_dma_channel_config* *dma_channel_config_t*

DMA channel trigger.

```
typedef struct _dma_transfer_config dma_transfer_config_t
```

DMA transfer configuration.

```
typedef void (*dma_callback)(struct _dma_handle *handle, void *userData, bool transferDone,  
uint32_t intmode)
```

Define Callback function for DMA.

```
typedef struct _dma_handle dma_handle_t
```

DMA transfer handle structure.

```
DMA_MAX_TRANSFER_COUNT
```

DMA max transfer size.

```
FSL_FEATURE_DMA_NUMBER_OF_CHANNELSn(x)
```

DMA channel numbers.

```
FSL_FEATURE_DMA_MAX_CHANNELS
```

```
FSL_FEATURE_DMA_ALL_CHANNELS
```

```
FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE
```

DMA head link descriptor table align size.

```
DMA_ALLOCATE_HEAD_DESCRIPTOR(name, number)
```

DMA head descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_HEAD_DESCRIPTOR_AT_NONCACHEABLE(name, number)
```

DMA head descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_LINK_DESCRIPTOR(name, number)
```

DMA link descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_LINK_DESCRIPTOR_AT_NONCACHEABLE(name, number)
```

DMA link descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_DATA_TRANSFER_BUFFER(name, width)

DMA transfer buffer address need to align with the transfer width.

DMA_CHANNEL_GROUP(channel)

DMA_CHANNEL_INDEX(base, channel)

DMA_COMMON_REG_GET(base, channel, reg)

DMA linked descriptor address align size.

DMA_COMMON_CONST_REG_GET(base, channel, reg)

DMA_COMMON_REG_SET(base, channel, reg, value)

DMA_DESCRIPTOR_END_ADDRESS(start, inc, bytes, width)

DMA descriptor end address calculate.

Parameters

- start – start address
- inc – address interleave size
- bytes – transfer bytes
- width – transfer width

DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width, srcInc, dstInc, bytes)

struct __dma_descriptor

#include <fsl_dma.h> DMA descriptor structure.

Public Members

volatile uint32_t xfercfg

Transfer configuration

void *srcEndAddr

Last source address of DMA transfer

void *dstEndAddr

Last destination address of DMA transfer

void *linkToNextDesc

Address of next DMA descriptor in chain

struct __dma_xfercfg

#include <fsl_dma.h> DMA transfer configuration.

Public Members

bool valid

Descriptor is ready to transfer

bool reload

Reload channel configuration register after current descriptor is exhausted

bool swtrig

Perform software trigger. Transfer if fired when 'valid' is set

bool clrtrig

Clear trigger

bool intA

Raises IRQ when transfer is done and set IRQA status register flag

bool intB

Raises IRQ when transfer is done and set IRQB status register flag

uint8_t byteWidth

Byte width of data to transfer

uint8_t srcInc

Increment source address by 'srcInc' x 'byteWidth'

uint8_t dstInc

Increment destination address by 'dstInc' x 'byteWidth'

uint16_t transferCount

Number of transfers

struct __dma_channel_trigger

#include <fsl_dma.h> DMA channel trigger.

Public Members

dma_trigger_type_t type

Select hardware trigger as edge triggered or level triggered.

dma_trigger_burst_t burst

Select whether hardware triggers cause a single or burst transfer.

dma_burst_wrap_t wrap

Select wrap type, source wrap or dest wrap, or both.

struct __dma_channel_config

#include <fsl_dma.h> DMA channel trigger.

Public Members

void *srcStartAddr

Source data address

void *dstStartAddr

Destination data address

void *nextDesc

Chain custom descriptor

uint32_t xferCfg

channel transfer configurations

dma_channel_trigger_t *trigger

DMA trigger type

bool isPeriph

select the request type

struct __dma_transfer_config

#include <fsl_dma.h> DMA transfer configuration.

Public Members

uint8_t *srcAddr

Source data address

uint8_t *dstAddr

Destination data address

uint8_t *nextDesc

Chain custom descriptor

dma_xfercfg_t xfercfg

Transfer options

bool isPeriph

DMA transfer is driven by peripheral

struct _dma_handle

#include <fsl_dma.h> DMA transfer handle structure.

Public Members

dma_callback callback

Callback function. Invoked when transfer of descriptor with interrupt flag finishes

void *userData

Callback function parameter

DMA_Type *base

DMA peripheral base address

uint8_t channel

DMA channel number

2.9 FLEXCOMM: FLEXCOMM Driver

2.10 FLEXCOMM Driver

FSL_FLEXCOMM_DRIVER_VERSION

FlexCOMM driver version 2.0.2.

enum FLEXCOMM_PERIPH_T

FLEXCOMM peripheral modes.

Values:

enumerator FLEXCOMM_PERIPH_NONE

No peripheral

enumerator FLEXCOMM_PERIPH_USART

USART peripheral

enumerator FLEXCOMM_PERIPH_SPI

SPI Peripheral

enumerator FLEXCOMM_PERIPH_I2C

I2C Peripheral

enumerator FLEXCOMM_PERIPH_I2S_TX

I2S TX Peripheral

enumerator FLEXCOMM_PERIPH_I2S_RX

I2S RX Peripheral

typedef void (*flexcomm_irq_handler_t)(void *base, void *handle)

Typedef for interrupt handler.

IRQn_Type const kFlexcommIrqs[]

Array with IRQ number for each FLEXCOMM module.

uint32_t FLEXCOMM_GetInstance(void *base)

Returns instance number for FLEXCOMM module with given base address.

status_t FLEXCOMM_Init(void *base, FLEXCOMM_PERIPH_T periph)

Initializes FLEXCOMM and selects peripheral mode according to the second parameter.

void FLEXCOMM_SetIRQHandler(void *base, flexcomm_irq_handler_t handler, void *flexcommHandle)

Sets IRQ handler for given FLEXCOMM module. It is used by drivers register IRQ handler according to FLEXCOMM mode.

2.11 GINT: Group GPIO Input Interrupt Driver

FSL_GINT_DRIVER_VERSION

Driver version.

enum _gint_comb

GINT combine inputs type.

Values:

enumerator kGINT_CombineOr

A grouped interrupt is generated when any one of the enabled inputs is active

enumerator kGINT_CombineAnd

A grouped interrupt is generated when all enabled inputs are active

enum _gint_trig

GINT trigger type.

Values:

enumerator kGINT_TrigEdge

Edge triggered based on polarity

enumerator kGINT_TrigLevel

Level triggered based on polarity

enum _gint_port

Values:

enumerator kGINT_Port0

enumerator kGINT_Port1

typedef enum _gint_comb gint_comb_t

GINT combine inputs type.

```
typedef enum _gint_trig gint_trig_t
```

GINT trigger type.

```
typedef enum _gint_port gint_port_t
```

```
typedef void (*gint_cb_t)(void)
```

GINT Callback function.

```
void GINT_Init(GINT_Type *base)
```

Initialize GINT peripheral.

This function initializes the GINT peripheral and enables the clock.

Parameters

- *base* – Base address of the GINT peripheral.

Return values

None. –

```
void GINT_SetCtrl(GINT_Type *base, gint_comb_t comb, gint_trig_t trig, gint_cb_t callback)
```

Setup GINT peripheral control parameters.

This function sets the control parameters of GINT peripheral.

Parameters

- *base* – Base address of the GINT peripheral.
- *comb* – Controls if the enabled inputs are logically ORed or ANDed for interrupt generation.
- *trig* – Controls if the enabled inputs are level or edge sensitive based on polarity.
- *callback* – This function is called when configured group interrupt is generated.

Return values

None. –

```
void GINT_GetCtrl(GINT_Type *base, gint_comb_t *comb, gint_trig_t *trig, gint_cb_t *callback)
```

Get GINT peripheral control parameters.

This function returns the control parameters of GINT peripheral.

Parameters

- *base* – Base address of the GINT peripheral.
- *comb* – Pointer to store combine input value.
- *trig* – Pointer to store trigger value.
- *callback* – Pointer to store callback function.

Return values

None. –

```
void GINT_ConfigPins(GINT_Type *base, gint_port_t port, uint32_t polarityMask, uint32_t enableMask)
```

Configure GINT peripheral pins.

This function enables and controls the polarity of enabled pin(s) of a given port.

Parameters

- *base* – Base address of the GINT peripheral.
- *port* – Port number.

- `polarityMask` – Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- `enableMask` – Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT__GetConfigPins(GINT_Type *base, gint_port_t port, uint32_t *polarityMask, uint32_t *enableMask)
```

Get GINT peripheral pin configuration.

This function returns the pin configuration of a given port.

Parameters

- `base` – Base address of the GINT peripheral.
- `port` – Port number.
- `polarityMask` – Pointer to store the polarity mask Each bit position indicates the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- `enableMask` – Pointer to store the enable mask. Each bit position indicates if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT__EnableCallback(GINT_Type *base)
```

Enable callback.

This function enables the interrupt for the selected GINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

- `base` – Base address of the GINT peripheral.

Return values

None. –

```
void GINT__DisableCallback(GINT_Type *base)
```

Disable callback.

This function disables the interrupt for the selected GINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

- `base` – Base address of the peripheral.

Return values

None. –

```
static inline void GINT__ClrStatus(GINT_Type *base)
```

Clear GINT status.

This function clears the GINT status bit.

Parameters

- `base` – Base address of the GINT peripheral.

Return values

None. –


```
static inline uint32_t GINT_GetStatus(GINT_Type *base)
```

Get GINT status.

This function returns the GINT status.

Parameters

- *base* – Base address of the GINT peripheral.

Return values

status – = 0 No group interrupt request. = 1 Group interrupt request active.

```
void GINT_Deinit(GINT_Type *base)
```

Deinitialize GINT peripheral.

This function disables the GINT clock.

Parameters

- *base* – Base address of the GINT peripheral.

Return values

None. –

2.12 Hashcrypt: The Cryptographic Accelerator

2.13 Hashcrypt Background HASH

```
void HASHCRYPT_SHA_SetCallback(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx,
                              hashcrypt_callback_t callback, void *userData)
```

Initializes the HASHCRYPT handle for background hashing.

This function initializes the hash context for background hashing (Non-blocking) APIs. This is less typical interface to hash function, but can be used for parallel processing, when main CPU has something else to do. Example is digital signature RSASSA-PKCS1-V1_5-VERIFY((n,e),M,S) algorithm, where background hashing of M can be started, then CPU can compute $S^e \bmod n$ (in parallel with background hashing) and once the digest becomes available, CPU can proceed to comparison of EM with EM'.

Parameters

- *base* – HASHCRYPT peripheral base address.
- *ctx* – **[out]** Hash context.
- *callback* – Callback function.
- *userData* – User data (to be passed as an argument to callback function, once callback is invoked from isr).

```
status_t HASHCRYPT_SHA_UpdateNonBlocking(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t
                                         *ctx, const uint8_t *input, size_t inputSize)
```

Create running hash on given data.

Configures the HASHCRYPT to compute new running hash as AHB master and returns immediately. HASHCRYPT AHB Master mode supports only aligned input address and can be called only once per continuous block of data. Every call to this function must be preceded with HASHCRYPT_SHA_Init() and finished with HASHCRYPT_SHA_Finish(). Once callback function is invoked by HASHCRYPT isr, it should set a flag for the main application to finalize the hashing (padding) and to read out the final digest by calling HASHCRYPT_SHA_Finish().

Parameters

- `base` – HASHCRYPT peripheral base address
- `ctx` – Specifies callback. Last incomplete 512-bit block of the input is copied into clear buffer for padding.
- `input` – 32-bit word aligned pointer to Input data.
- `inputSize` – Size of input data in bytes (must be word aligned)

Returns

Status of the hash update operation.

2.14 Hashcrypt common functions

FSL_HASHCRYPT_DRIVER_VERSION

HASHCRYPT driver version. Version 2.2.16.

Current version: 2.2.16

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Support loading AES key from unaligned address
- Version 2.0.2
 - Support loading AES key from unaligned address for different compiler and core variants
- Version 2.0.3
 - Remove SHA512 and AES ICB algorithm definitions
- Version 2.0.4
 - Add SHA context switch support
- Version 2.1.0
 - Update the register name and macro to align with new header.
- Version 2.1.1
 - Fix MISRA C-2012.
- Version 2.1.2
 - Support loading AES input data from unaligned address.
- Version 2.1.3
 - Fix MISRA C-2012.
- Version 2.1.4
 - Fix context switch cannot work when switching from AES.
- Version 2.1.5
 - Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` to prevent possible optimization issue.
- Version 2.2.0
 - Add AES-OFB and AES-CFB mixed IP/SW modes.

- Version 2.2.1
 - Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` prevent compiler from reordering memory write when `-O2` or higher is used.
- Version 2.2.2
 - Add data synchronization barrier inside `hashcrypt_sha_ldm_stm_16_words()` to fix optimization issue
- Version 2.2.3
 - Added check for size in `hashcrypt_aes_one_block` to prevent overflowing COUNT field in MEMCTRL register, if its bigger than COUNT field do a multiple runs.
- Version 2.2.4
 - In all `HASHCRYPT_AES_xx` functions have been added setting CTRL_MODE bitfield to 0 after processing data, which decreases power consumption.
- Version 2.2.5
 - Add data synchronization barrier and instruction synchronization barrier inside `hashcrypt_sha_process_message_data()` to fix optimization issue
- Version 2.2.6
 - Add data synchronization barrier inside `HASHCRYPT_SHA_Update()` and `hashcrypt_get_data()` function to fix optimization issue on MDK and ARMGCC release targets
- Version 2.2.7
 - Add data synchronization barrier inside `HASHCRYPT_SHA_Update()` to fix optimization issue on MCUX IDE release target
- Version 2.2.8
 - Unify hashcrypt hashing behavior between aligned and unaligned input data
- Version 2.2.9
 - Add handling of set ERROR bit in the STATUS register
- Version 2.2.10
 - Fix missing error statement in `hashcrypt_save_running_hash()`
- Version 2.2.11
 - Fix incorrect SHA-256 calculation for long messages with reload
- Version 2.2.12
 - Fix hardfault issue on the Keil compiler due to unaligned `memcpy()` input on some optimization levels
- Version 2.2.13
 - Added function `hashcrypt_seed_prng()` which loading random number into PRNG_SEED register before AES operation for SCA protection
- Version 2.2.14
 - Modify function `hashcrypt_get_data()` to prevent issue with unaligned access
- Version 2.2.15
 - Add wait on DIGEST BIT inside `hashcrypt_sha_one_block()` to fix issues with some optimization flags
- Version 2.2.16

- Add DSB instruction inside `hashcrypt_sha_ldm_stm_16_words()` to fix issues with some optimization flags

`enum _hashcrypt_algo_t`

Algorithm used for Hashcrypt operation.

Values:

enumerator `kHASHCRYPT_Sha1`

SHA_1

enumerator `kHASHCRYPT_Sha256`

SHA_256

enumerator `kHASHCRYPT_Aes`

AES

`typedef enum _hashcrypt_algo_t hashcrypt_algo_t`

Algorithm used for Hashcrypt operation.

`void HASHCRYPT_Init(HASHCRYPT_Type *base)`

Enables clock and disables reset for HASHCRYPT peripheral.

Enable clock and disable reset for HASHCRYPT.

Parameters

- `base` – HASHCRYPT base address

`void HASHCRYPT_Deinit(HASHCRYPT_Type *base)`

Disables clock for HASHCRYPT peripheral.

Disable clock and enable reset.

Parameters

- `base` – HASHCRYPT base address

`HASHCRYPT_MODE_SHA1`

Algorithm definitions correspond with the values for Mode field in Control register !

`HASHCRYPT_MODE_SHA256`

`HASHCRYPT_MODE_AES`

2.15 Hashcrypt AES

`enum _hashcrypt_aes_mode_t`

AES mode.

Values:

enumerator `kHASHCRYPT_AesEcb`

AES ECB mode

enumerator `kHASHCRYPT_AesCbc`

AES CBC mode

enumerator `kHASHCRYPT_AesCtr`

AES CTR mode

enum `_hashcrypt_aes_keysize_t`

Size of AES key.

Values:

enumerator `kHASHCRYPT_Aes128`

AES 128 bit key

enumerator `kHASHCRYPT_Aes192`

AES 192 bit key

enumerator `kHASHCRYPT_Aes256`

AES 256 bit key

enumerator `kHASHCRYPT_InvalidKey`

AES invalid key

enum `_hashcrypt_key`

HASHCRYPT key source selection.

Values:

enumerator `kHASHCRYPT_UserKey`

HASHCRYPT user key

enumerator `kHASHCRYPT_SecretKey`

HASHCRYPT secret key (dedicated hw bus from PUF)

typedef enum `_hashcrypt_aes_mode_t` `hashcrypt_aes_mode_t`

AES mode.

typedef enum `_hashcrypt_aes_keysize_t` `hashcrypt_aes_keysize_t`

Size of AES key.

typedef enum `_hashcrypt_key` `hashcrypt_key_t`

HASHCRYPT key source selection.

typedef struct `_hashcrypt_handle` `hashcrypt_handle_t`

struct `_hashcrypt_handle` `__attribute__((aligned))`

`status_t` `HASHCRYPT_AES_SetKey(HASHCRYPT_Type *base, hashcrypt_handle_t *handle, const uint8_t *key, size_t keySize)`

Set AES key to `hashcrypt_handle_t` struct and optionally to HASHCRYPT.

Sets the AES key for encryption/decryption with the `hashcrypt_handle_t` structure. The `hashcrypt_handle_t` input argument specifies key source.

Parameters

- `base` – HASHCRYPT peripheral base address.
- `handle` – Handle used for the request.
- `key` – 0-mod-4 aligned pointer to AES key.
- `keySize` – AES key size in bytes. Shall equal 16, 24 or 32.

Returns

status from set key operation

`status_t` `HASHCRYPT_AES_EncryptEcb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, size_t size)`

Encrypts AES on one or multiple 128-bit block(s).

Encrypts AES. The source plaintext and destination ciphertext can overlap in system memory.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptEcb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                   const uint8_t *ciphertext, uint8_t *plaintext, size_t  
                                   size)
```

Decrypts AES on one or multiple 128-bit block(s).

Decrypts AES. The source ciphertext and destination plaintext can overlap in system memory.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- ciphertext – Input plain text to encrypt
- plaintext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

Returns

Status from decrypt operation

```
status_t HASHCRYPT_AES_EncryptCbc(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                   const uint8_t *plaintext, uint8_t *ciphertext, size_t  
                                   size, const uint8_t iv[16])
```

Encrypts AES using CBC block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptCbc(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                   const uint8_t *ciphertext, uint8_t *plaintext, size_t  
                                   size, const uint8_t iv[16])
```

Decrypts AES using CBC block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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.

Returns

Status from decrypt operation

```
status_t HASHCRYPT_AES_CryptCtr(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                const uint8_t *input, uint8_t *output, size_t size, uint8_t  
                                counter[16U], uint8_t counterlast[16U], size_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 – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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)
- counterlast – **[out]** Output cipher of last counter, for chained CTR calls (statefull encryption). 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 HASHCRYPT_AES_CryptOfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                const uint8_t *input, uint8_t *output, size_t size, const  
                                uint8_t iv[16U])
```

Encrypts or decrypts AES using OFB block mode.

Encrypts or decrypts AES using OFB block mode. AES OFB 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 – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- input – Input data for OFB block mode

- output – **[out]** Output data for OFB block mode
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_EncryptCfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                  const uint8_t *plaintext, uint8_t *ciphertext, size_t size,  
                                  const uint8_t iv[16])
```

Encrypts AES using CFB block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptCfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                  const uint8_t *ciphertext, uint8_t *plaintext, size_t size,  
                                  const uint8_t iv[16])
```

Decrypts AES using CFB block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plaintext 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.

Returns

Status from encrypt operation

`HASHCRYPT_AES_BLOCK_SIZE`

AES block size in bytes

`AES_ENCRYPT`

`AES_DECRYPT`

`struct _hashcrypt_handle`

`#include <fsl_hashcrypt.h>` Specify HASHCRYPT's key resource.

Public Members

`uint32_t keyWord[8]`

Copy of user key (set by `HASHCRYPT_AES_SetKey()`).

hashcrypt_key_t keyType

For operations with key (such as AES encryption/decryption), specify key type.

2.16 Hashcrypt HASH

`typedef struct _hashcrypt_hash_ctx_t hashcrypt_hash_ctx_t`

Storage type used to save hash context.

`typedef void (*hashcrypt_callback_t)(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, status_t status, void *userData)`

HASHCRYPT background hash callback function.

`status_t HASHCRYPT_SHA(HASHCRYPT_Type *base, hashcrypt_algo_t algo, const uint8_t *input, size_t inputSize, uint8_t *output, size_t *outputSize)`

Create HASH on given data.

Perform the full SHA in one function call. The function is blocking.

Parameters

- base – HASHCRYPT peripheral base address
- algo – Underlying algorithm to use for hash computation.
- input – Input data
- inputSize – Size of input data 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.

`status_t HASHCRYPT_SHA_Init(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, hashcrypt_algo_t algo)`

Initialize HASH context.

This function initializes the HASH.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[out]** Output hash context
- algo – Underlying algorithm to use for hash computation.

Returns

Status of initialization

`status_t HASHCRYPT_SHA_Update(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, const uint8_t *input, size_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. The function blocks. If it returns kStatus_Success, the running hash has been updated (HASHCRYPT has processed the input data), so the memory at `input` pointer can be released back to system. The HASHCRYPT context buffer is updated with the running hash and with all necessary information to support possible context switch.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** HASH context
- input – Input data
- inputSize – Size of input data in bytes

Returns

Status of the hash update operation

```
status_t HASHCRYPT_SHA_Finish(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, uint8_t  
                             *output, size_t *outputSize)
```

Finalize hashing.

Outputs the final hash (computed by HASHCRYPT_HASH_Update()) and erases the context.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** Input hash context
- output – **[out]** Output hash data
- outputSize – **[inout]** Optional parameter (can be passed as NULL). On function entry, it specifies the size of output[] buffer. On function return, it stores the number of updated output bytes.

Returns

Status of the hash finish operation

HASHCRYPT_HASH_CTX_SIZE

HASHCRYPT HASH Context size.

```
struct _hashcrypt_hash_ctx_t
```

#include <fsl_hashcrypt.h> Storage type used to save hash context.

Public Members

uint32_t x[30]

storage

2.17 I2C: Inter-Integrated Circuit Driver

2.18 I2C DMA Driver

```
void I2C_MasterTransferCreateHandleDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                                       i2c_master_dma_transfer_callback_t callback, void  
                                       *userData, dma_handle_t *dmaHandle)
```

Init the I2C handle which is used in transactional functions.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- callback – pointer to user callback function
- userData – user param passed to the callback function
- dmaHandle – DMA handle pointer

```
status_t I2C_MasterTransferDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,
                              i2c_master_transfer_t *xfer)
```

Performs a master dma non-blocking transfer on the I2C bus.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- xfer – pointer to transfer structure of i2c_master_transfer_t

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_Busy – Previous transmission still not finished.
- kStatus_I2C_Timeout – Transfer error, wait signal timeout.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive Nak during transfer.

```
status_t I2C_MasterTransferGetCountDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,
                                       size_t *count)
```

Get master transfer status during a dma non-blocking transfer.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- count – Number of bytes transferred so far by the non-blocking transaction.

```
void I2C_MasterTransferAbortDMA(I2C_Type *base, i2c_master_dma_handle_t *handle)
```

Abort a master dma non-blocking transfer in a early time.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure

```
FSL_I2C_DMA_DRIVER_VERSION
```

I2C DMA driver version.

```
typedef struct _i2c_master_dma_handle i2c_master_dma_handle_t
```

I2C master dma handle typedef.

```
typedef void (*i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t
*handle, status_t status, void *userData)
```

I2C master dma transfer callback typedef.

```
typedef void (*flexcomm_i2c_dma_master_irq_handler_t)(I2C_Type *base,
i2c_master_dma_handle_t *handle)
```

Typedef for master dma handler.

```
I2C_MAX_DMA_TRANSFER_COUNT
```

Maximum length of single DMA transfer (determined by capability of the DMA engine)

```
struct _i2c_master_dma_handle
```

#include <fsl_i2c_dma.h> I2C master dma transfer structure.

Public Members

uint8_t state

Transfer state machine current state.

uint32_t transferCount

Indicates progress of the transfer

uint32_t remainingBytesDMA

Remaining byte count to be transferred using DMA.

uint8_t *buf

Buffer pointer for current state.

bool checkAddrNack

Whether to check the nack signal is detected during addressing.

dma_handle_t *dmaHandle

The DMA handler used.

i2c_master_transfer_t transfer

Copy of the current transfer info.

i2c_master_dma_transfer_callback_t completionCallback

Callback function called after dma transfer finished.

void *userData

Callback parameter passed to callback function.

2.19 I2C Driver

FSL_I2C_DRIVER_VERSION

I2C driver version.

I2C status return codes.

Values:

enumerator kStatus_I2C_Busy

The master is already performing a transfer.

enumerator kStatus_I2C_Idle

The slave driver is idle.

enumerator kStatus_I2C_Nak

The slave device sent a NAK in response to a byte.

enumerator kStatus_I2C_InvalidParameter

Unable to proceed due to invalid parameter.

enumerator kStatus_I2C_BitError

Transferred bit was not seen on the bus.

enumerator kStatus_I2C_ArbitrationLost

Arbitration lost error.

enumerator kStatus_I2C_NoTransferInProgress

Attempt to abort a transfer when one is not in progress.

enumerator kStatus_I2C_DmaRequestFail

DMA request failed.

enumerator kStatus_I2C_StartStopError

Start and stop error.

enumerator kStatus_I2C_UnexpectedState

Unexpected state.

enumerator kStatus_I2C_Timeout

Timeout when waiting for I2C master/slave pending status to set to continue transfer.

enumerator kStatus_I2C_Addr_Nak

NAK received for Address

enumerator kStatus_I2C_EventTimeout

Timeout waiting for bus event.

enumerator kStatus_I2C_SclLowTimeout

Timeout SCL signal remains low.

enum _i2c_status_flags

I2C status flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingFlag

The I2C module is waiting for software interaction. bit 0

enumerator kI2C_MasterArbitrationLostFlag

The arbitration of the bus was lost. There was collision on the bus. bit 4

enumerator kI2C_MasterStartStopErrorFlag

There was an error during start or stop phase of the transaction. bit 6

enumerator kI2C_MasterIdleFlag

The I2C master idle status. bit 5

enumerator kI2C_MasterRxReadyFlag

The I2C master rx ready status. bit 1

enumerator kI2C_MasterTxReadyFlag

The I2C master tx ready status. bit 2

enumerator kI2C_MasterAddrNackFlag

The I2C master address nack status. bit 7

enumerator kI2C_MasterDataNackFlag

The I2C master data nack status. bit 3

enumerator kI2C_SlavePendingFlag

The I2C module is waiting for software interaction. bit 8

enumerator kI2C_SlaveNotStretching

Indicates whether the slave is currently stretching clock (0 = yes, 1 = no). bit 11

enumerator kI2C_SlaveSelected

Indicates whether the slave is selected by an address match. bit 14

enumerator kI2C_SaveDeselected

Indicates that slave was previously deselected (deselect event took place, w1c). bit 15

enumerator kI2C_SlaveAddressedFlag

One of the I2C slave's 4 addresses is matched. bit 22

enumerator kI2C_SlaveReceiveFlag

Slave receive data available. bit 9

enumerator kI2C_SlaveTransmitFlag

Slave data can be transmitted. bit 10

enumerator kI2C_SlaveAddress0MatchFlag

Slave address0 match. bit 20

enumerator kI2C_SlaveAddress1MatchFlag

Slave address1 match. bit 12

enumerator kI2C_SlaveAddress2MatchFlag

Slave address2 match. bit 13

enumerator kI2C_SlaveAddress3MatchFlag

Slave address3 match. bit 21

enumerator kI2C_MonitorReadyFlag

The I2C monitor ready interrupt. bit 16

enumerator kI2C_MonitorOverflowFlag

The monitor data overrun interrupt. bit 17

enumerator kI2C_MonitorActiveFlag

The monitor is active. bit 18

enumerator kI2C_MonitorIdleFlag

The monitor idle interrupt. bit 19

enumerator kI2C_EventTimeoutFlag

The bus event timeout interrupt. bit 24

enumerator kI2C_SclTimeoutFlag

The SCL timeout interrupt. bit 25

enumerator kI2C_MasterAllClearFlags

enumerator kI2C_SlaveAllClearFlags

enumerator kI2C_CommonAllClearFlags

enum _i2c_interrupt_enable

I2C interrupt enable.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingInterruptEnable

The I2C master communication pending interrupt.

enumerator kI2C_MasterArbitrationLostInterruptEnable

The I2C master arbitration lost interrupt.

enumerator kI2C_MasterStartStopErrorInterruptEnable

The I2C master start/stop timing error interrupt.

enumerator kI2C_SlavePendingInterruptEnable

The I2C slave communication pending interrupt.

enumerator kI2C_SlaveNotStretchingInterruptEnable

The I2C slave not stretching interrupt, deep-sleep mode can be entered only when this interrupt occurs.

enumerator kI2C_SlaveDeselectedInterruptEnable

The I2C slave deselection interrupt.

enumerator kI2C_MonitorReadyInterruptEnable

The I2C monitor ready interrupt.

enumerator kI2C_MonitorOverflowInterruptEnable

The monitor data overrun interrupt.

enumerator kI2C_MonitorIdleInterruptEnable

The monitor idle interrupt.

enumerator kI2C_EventTimeoutInterruptEnable

The bus event timeout interrupt.

enumerator kI2C_SclTimeoutInterruptEnable

The SCL timeout interrupt.

enumerator kI2C_MasterAllInterruptEnable

enumerator kI2C_SlaveAllInterruptEnable

enumerator kI2C_CommonAllInterruptEnable

I2C_RETRY_TIMES

Retry times for waiting flag.

I2C_MASTER_TRANSMIT_IGNORE_LAST_NACK

Whether to ignore the nack signal of the last byte during master transmit.

I2C_STAT_MSTCODE_IDLE

Master Idle State Code

I2C_STAT_MSTCODE_RXREADY

Master Receive Ready State Code

I2C_STAT_MSTCODE_TXREADY

Master Transmit Ready State Code

I2C_STAT_MSTCODE_NACKADR

Master NACK by slave on address State Code

I2C_STAT_MSTCODE_NACKDAT

Master NACK by slave on data State Code

I2C_STAT_SLVST_ADDR

I2C_STAT_SLVST_RX

I2C_STAT_SLVST_TX

2.20 I2C Master Driver

`void I2C_MasterGetDefaultConfig(i2c_master_config_t *masterConfig)`

Provides a default configuration for the I2C master peripheral.

This function provides the following default configuration for the I2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->baudRate__Bps     = 100000U;
masterConfig->enableTimeout     = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i2c_master_config_t`.

`void I2C_MasterInit(I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)`

Initializes the I2C master peripheral.

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `srcClock_Hz` – Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`void I2C_MasterDeinit(I2C_Type *base)`

Deinitializes the I2C master peripheral.

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

`uint32_t I2C_GetInstance(I2C_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 I2C peripheral base address.

Returns

I2C instance number starting from 0.

`static inline void I2C_MasterReset(I2C_Type *base)`

Performs a software reset.

Restores the I2C master peripheral to reset conditions.

Parameters

- base – The I2C peripheral base address.

static inline void I2C_MasterEnable(I2C_Type *base, bool enable)

Enables or disables the I2C module as master.

Parameters

- base – The I2C peripheral base address.
- enable – Pass true to enable or false to disable the specified I2C as master.

uint32_t I2C_GetStatusFlags(I2C_Type *base)

Gets the I2C status flags.

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i2c_status_flags`.

Parameters

- base – The I2C 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 I2C_ClearStatusFlags(I2C_Type *base, uint32_t statusMask)

Clears the I2C status flag state.

Refer to `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags` to see the clearable flags. Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`, `_i2c_master_status_flags` and `_i2c_slave_status_flags`.

Parameters

- base – The I2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of the members in `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags`. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)

Clears the I2C master status flag state.

Deprecated:

Do not use this function. It has been superseded by `I2C_ClearStatusFlags`. The following status register flags can be cleared:

- `kI2C_MasterArbitrationLostFlag`
- `kI2C_MasterStartStopErrorFlag`

Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_status_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

```
static inline void I2C_EnableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Enables the I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I2C_DisableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Disables the I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I2C_GetEnabledInterrupts(I2C_Type *base)
```

Returns the set of currently enabled I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.

Returns

A bitmask composed of `_i2c_interrupt_enable` enumerators OR'd together to indicate the set of enabled interrupts.

```
void I2C_MasterSetBaudRate(I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
```

Sets the I2C bus frequency for master transactions.

The I2C 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 I2C peripheral base address.
- `srcClock_Hz` – I2C functional clock frequency in Hertz.
- `baudRate_Bps` – Requested bus frequency in bits per second.

```
void I2C_MasterSetTimeoutValue(I2C_Type *base, uint8_t timeout_Ms, uint32_t srcClock_Hz)
```

Sets the I2C bus timeout value.

If the SCL signal remains low or bus does not have event longer than the timeout value, `kI2C_SclTimeoutFlag` or `kI2C_EventTimeoutFlag` is set. This can indicate the bus is held by slave or any fault occurs to the I2C module.

Parameters

- `base` – The I2C peripheral base address.
- `timeout_Ms` – Timeout value in millisecond.
- `srcClock_Hz` – I2C functional clock frequency in Hertz.

`static inline bool I2C_MasterGetBusIdleState(I2C_Type *base)`

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The I2C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

`status_t I2C_MasterStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)`

Sends a START on the I2C bus.

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

- `base` – I2C peripheral base pointer
- `address` – 7-bit slave device address.
- `direction` – Master transfer directions(transmit/receive).

Return values

- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy.

`status_t I2C_MasterStop(I2C_Type *base)`

Sends a STOP signal on the I2C bus.

Return values

- `kStatus_Success` – Successfully send the stop signal.
- `kStatus_I2C_Timeout` – Send stop signal failed, timeout.

`static inline status_t I2C_MasterRepeatedStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)`

Sends a REPEATED START on the I2C bus.

Parameters

- `base` – I2C peripheral base pointer
- `address` – 7-bit slave device address.
- `direction` – Master transfer directions(transmit/receive).

Return values

- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy but not occupied by current I2C master.

status_t I2C_MasterWriteBlocking(I2C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)

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_I2C_Nak*.

Parameters

- *base* – The I2C peripheral base address.
- *txBuff* – The pointer to the data to be transferred.
- *txSize* – The length in bytes of the data to be transferred.
- *flags* – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use *kI2C_TransferDefaultFlag*

Return values

- *kStatus__Success* – Data was sent successfully.
- *kStatus_I2C_Busy* – Another master is currently utilizing the bus.
- *kStatus_I2C_Nak* – The slave device sent a NAK in response to a byte.
- *kStatus_I2C_ArbitrationLost* – Arbitration lost error.

status_t I2C_MasterReadBlocking(I2C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transfer on the I2C bus.

Parameters

- *base* – The I2C peripheral base address.
- *rxBuff* – The pointer to the data to be transferred.
- *rxSize* – The length in bytes of the data to be transferred.
- *flags* – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use *kI2C_TransferDefaultFlag*

Return values

- *kStatus__Success* – Data was received successfully.
- *kStatus_I2C_Busy* – Another master is currently utilizing the bus.
- *kStatus_I2C_Nak* – The slave device sent a NAK in response to a byte.
- *kStatus_I2C_ArbitrationLost* – Arbitration lost error.

status_t I2C_MasterTransferBlocking(I2C_Type *base, *i2c_master_transfer_t* *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

- *base* – I2C peripheral base address.
- *xfer* – Pointer to the transfer structure.

Return values

- *kStatus__Success* – Successfully complete the data transmission.
- *kStatus_I2C_Busy* – Previous transmission still not finished.

- `kStatus_I2C_Timeout` – Transfer error, wait signal timeout.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.
- `kStatus_I2C_Addr_Nak` – Transfer error, receive NAK during addressing.

```
void I2C_MasterTransferCreateHandle(I2C_Type *base, i2c_master_handle_t *handle,  
                                   i2c_master_transfer_callback_t callback, void *userData)
```

Creates a new handle for the I2C 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 `I2C_MasterTransferAbort()` API shall be called.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – **[out]** Pointer to the I2C master driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

```
status_t I2C_MasterTransferNonBlocking(I2C_Type *base, i2c_master_handle_t *handle,  
                                       i2c_master_transfer_t *xfer)
```

Performs a non-blocking transaction on the I2C bus.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.
- `xfer` – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I2C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

```
status_t I2C_MasterTransferGetCount(I2C_Type *base, i2c_master_handle_t *handle, size_t  
                                   *count)
```

Returns number of bytes transferred so far.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.
- `count` – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_Success` –
- `kStatus_I2C_Busy` –

```
status_t I2C_MasterTransferAbort(I2C_Type *base, i2c_master_handle_t *handle)
```

Terminates a non-blocking I2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

Parameters

- base – The I2C peripheral base address.
- handle – Pointer to the I2C master driver handle.

Return values

- kStatus_Success – A transaction was successfully aborted.
- kStatus_I2C_Timeout – Timeout during polling for flags.

void I2C_MasterTransferHandleIRQ(I2C_Type *base, i2c_master_handle_t *handle)
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 I2C peripheral base address.
- handle – Pointer to the I2C master driver handle.

enum _i2c_direction
Direction of master and slave transfers.

Values:

enumerator kI2C_Write
Master transmit.

enumerator kI2C_Read
Master receive.

enum _i2c_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 _i2c_master_transfer::flags field.

Values:

enumerator kI2C_TransferDefaultFlag
Transfer starts with a start signal, stops with a stop signal.

enumerator kI2C_TransferNoStartFlag
Don't send a start condition, address, and sub address

enumerator kI2C_TransferRepeatedStartFlag
Send a repeated start condition

enumerator kI2C_TransferNoStopFlag
Don't send a stop condition.

enum _i2c_transfer_states
States for the state machine used by transactional APIs.

Values:

enumerator kIdleState

enumerator kTransmitSubaddrState

enumerator kTransmitDataState

enumerator kReceiveDataBeginState

enumerator kReceiveDataState

enumerator kReceiveLastDataState

enumerator kStartState

enumerator kStopState

enumerator kWaitForCompletionState

typedef enum *_i2c_direction* i2c_direction_t

Direction of master and slave transfers.

typedef struct *_i2c_master_config* i2c_master_config_t

Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_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 *_i2c_master_transfer* i2c_master_transfer_t

I2C master transfer typedef.

typedef struct *_i2c_master_handle* i2c_master_handle_t

I2C master handle typedef.

typedef void (*i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t 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 I2C_MasterTransferCreateHandle().

Param base

The I2C peripheral base address.

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 *_i2c_master_config*

#include <fsl_i2c.h> Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_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.

uint32_t baudRate_Bps

Desired baud rate in bits per second.

bool enableTimeout

Enable internal timeout function.

uint8_t timeout_Ms

Event timeout and SCL low timeout value.

struct _i2c_master_transfer

#include <fsl_i2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_i2c_master_transfer_flags` for available options. Set to 0 or `kI2C_TransferDefaultFlag` for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i2c_direction_t direction

Either `kI2C_Read` or `kI2C_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 _i2c_master_handle

#include <fsl_i2c.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 transferCount

Indicates progress of the transfer

uint32_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

`bool checkAddrNack`

Whether to check the nack signal is detected during addressing.

`i2c_master_transfer_t transfer`

Copy of the current transfer info.

`i2c_master_transfer_callback_t completionCallback`

Callback function pointer.

`void *userData`

Application data passed to callback.

2.21 I2C Slave Driver

`void I2C_SlaveGetDefaultConfig(i2c_slave_config_t *slaveConfig)`

Provides a default configuration for the I2C slave peripheral.

This function provides the following default configuration for the I2C slave peripheral:

```
slaveConfig->enableSlave = true;
slaveConfig->address0.disable = false;
slaveConfig->address0.address = 0u;
slaveConfig->address1.disable = true;
slaveConfig->address2.disable = true;
slaveConfig->address3.disable = true;
slaveConfig->busSpeed = kI2C_SlaveStandardMode;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `I2C_SlaveInit()`. Be sure to override at least the `address0.address` 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 `i2c_slave_config_t`.

`status_t I2C_SlaveInit(I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t srcClock_Hz)`

Initializes the I2C slave peripheral.

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I2C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `srcClock_Hz` – Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

`void I2C_SlaveSetAddress(I2C_Type *base, i2c_slave_address_register_t addressRegister, uint8_t address, bool addressDisable)`

Configures Slave Address n register.

This function writes new value to Slave Address register.

Parameters

- `base` – The I2C peripheral base address.

- `addressRegister` – The module supports multiple address registers. The parameter determines which one shall be changed.
- `address` – The slave address to be stored to the address register for matching.
- `addressDisable` – Disable matching of the specified address register.

`void I2C_SlaveDeinit(I2C_Type *base)`

Deinitializes the I2C slave peripheral.

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

`static inline void I2C_SlaveEnable(I2C_Type *base, bool enable)`

Enables or disables the I2C module as slave.

Parameters

- `base` – The I2C peripheral base address.
- `enable` – True to enable or false to disable.

`static inline void I2C_SlaveClearStatusFlags(I2C_Type *base, uint32_t statusMask)`

Clears the I2C status flag state.

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

See also:

`_i2c_slave_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_SlaveGetStatusFlags()`.

`status_t I2C_SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)`

Performs a polling send transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- `base` – The I2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

`kStatus_Success` Data has been sent.

Returns

`kStatus_Fail` Unexpected slave state (master data write while master read from slave is expected).

status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- base – The I2C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been received.

Returns

kStatus_Fail Unexpected slave state (master data read while master write to slave is expected).

void I2C_SlaveTransferCreateHandle(I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)

Creates a new handle for the I2C 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 I2C_SlaveTransferAbort() API shall be called.

Parameters

- base – The I2C peripheral base address.
- handle – **[out]** Pointer to the I2C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and I2C_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 I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes kI2C_SlaveTransmitEvent callback. If no slave Rx transfer is busy, a master write to slave request invokes kI2C_SlaveReceiveEvent callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_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 kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The I2C peripheral base address.
- handle – Pointer to i2c_slave_handle_t structure which stores the transfer state.

- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`status_t I2C_SlaveSetSendBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, const void *txData, size_t txSize, uint32_t eventMask)`

Starts accepting master read from slave requests.

The function can be called in response to `kI2C_SlaveTransmitEvent` callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kI2C_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 `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `txData` – Pointer to data to send to master.
- `txSize` – Size of `txData` in bytes.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`status_t I2C_SlaveSetReceiveBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *rxData, size_t rxSize, uint32_t eventMask)`

Starts accepting master write to slave requests.

The function can be called in response to `kI2C_SlaveReceiveEvent` callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kI2C_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 `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.

- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `rxData` – Pointer to data to store data from master.
- `rxSize` – Size of `rxData` in bytes.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus__Success` – Slave transfers were successfully started.
- `kStatus__I2C__Busy` – Slave transfers have already been started on this handle.

```
static inline uint32_t I2C_SlaveGetReceivedAddress(I2C_Type *base, volatile i2c_slave_transfer_t *transfer)
```

Returns the slave address sent by the I2C master.

This function should only be called from the address match event callback `kI2C_SlaveAddressMatchEvent`.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

```
void I2C_SlaveTransferAbort(I2C_Type *base, i2c_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 I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

Return values

- `kStatus__Success` –
- `kStatus__I2C__Idle` –

```
status_t I2C_SlaveTransferGetCount(I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
```

Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus__InvalidArgument` – count is Invalid.
- `kStatus__Success` – Successfully return the count.

void I2C_SlaveTransferHandleIRQ(I2C_Type *base, i2c_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 I2C peripheral base address.
- handle – Pointer to i2c_slave_handle_t structure which stores the transfer state.

enum _i2c_slave_address_register

I2C slave address register.

Values:

enumerator kI2C_SlaveAddressRegister0

Slave Address 0 register.

enumerator kI2C_SlaveAddressRegister1

Slave Address 1 register.

enumerator kI2C_SlaveAddressRegister2

Slave Address 2 register.

enumerator kI2C_SlaveAddressRegister3

Slave Address 3 register.

enum _i2c_slave_address_qual_mode

I2C slave address match options.

Values:

enumerator kI2C_QualModeMask

The SLVQUAL0 field (qualAddress) is used as a logical mask for matching address0.

enumerator kI2C_QualModeExtend

The SLVQUAL0 (qualAddress) field is used to extend address 0 matching in a range of addresses.

enum _i2c_slave_bus_speed

I2C slave bus speed options.

Values:

enumerator kI2C_SlaveStandardMode

enumerator kI2C_SlaveFastMode

enumerator kI2C_SlaveFastModePlus

enumerator kI2C_SlaveHsMode

enum _i2c_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 I2C_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 `kI2C_SlaveAddressMatchEvent`

Received the slave address after a start or repeated start.

enumerator `kI2C_SlaveTransmitEvent`

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator `kI2C_SlaveReceiveEvent`

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator `kI2C_SlaveCompletionEvent`

All data in the active transfer have been consumed.

enumerator `kI2C_SlaveDeselectedEvent`

The slave function has become deselected (SLVSEL flag changing from 1 to 0).

enumerator `kI2C_SlaveAllEvents`

Bit mask of all available events.

enum `_i2c_slave_fsm`

I2C slave software finite state machine states.

Values:

enumerator `kI2C_SlaveFsmAddressMatch`

enumerator `kI2C_SlaveFsmReceive`

enumerator `kI2C_SlaveFsmTransmit`

typedef enum `_i2c_slave_address_register` `i2c_slave_address_register_t`

I2C slave address register.

typedef struct `_i2c_slave_address` `i2c_slave_address_t`

Data structure with 7-bit Slave address and Slave address disable.

typedef enum `_i2c_slave_address_qual_mode` `i2c_slave_address_qual_mode_t`

I2C slave address match options.

typedef enum `_i2c_slave_bus_speed` `i2c_slave_bus_speed_t`

I2C slave bus speed options.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`

Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the `I2C_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 `_i2c_slave_transfer_event` `i2c_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 `I2C_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 *_i2c_slave_handle* i2c_slave_handle_t
I2C slave handle typedef.

typedef struct *_i2c_slave_transfer* i2c_slave_transfer_t
I2C slave transfer structure.

typedef void (*i2c_slave_transfer_callback_t)(I2C_Type *base, volatile i2c_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 I2C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the I2C 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 enum *_i2c_slave_fsm* i2c_slave_fsm_t
I2C slave software finite state machine states.

typedef void (*flexcomm_i2c_master_irq_handler_t)(I2C_Type *base, i2c_master_handle_t *handle)
Typedef for master interrupt handler.

typedef void (*flexcomm_i2c_slave_irq_handler_t)(I2C_Type *base, i2c_slave_handle_t *handle)
Typedef for slave interrupt handler.

struct *_i2c_slave_address*
#include <fsl_i2c.h> Data structure with 7-bit Slave address and Slave address disable.

Public Members

uint8_t address
7-bit Slave address SLVADR.

bool addressDisable
Slave address disable SADISABLE.

struct *_i2c_slave_config*
#include <fsl_i2c.h> Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_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

i2c_slave_address_t address0
Slave's 7-bit address and disable.

i2c_slave_address_t address1

Alternate slave 7-bit address and disable.

i2c_slave_address_t address2

Alternate slave 7-bit address and disable.

i2c_slave_address_t address3

Alternate slave 7-bit address and disable.

i2c_slave_address_qual_mode_t qualMode

Qualify mode for slave address 0.

uint8_t qualAddress

Slave address qualifier for address 0.

i2c_slave_bus_speed_t busSpeed

Slave bus speed mode. If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The busSpeed value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the busSpeed mode is unknown at compile time, use the longest data setup time kI2C_SlaveStandardMode (250 ns)

bool enableSlave

Enable slave mode.

struct *_i2c_slave_transfer*

#include <fsl_i2c.h> I2C slave transfer structure.

Public Members

i2c_slave_handle_t *handle

Pointer to handle that contains this transfer.

i2c_slave_transfer_event_t event

Reason the callback is being invoked.

uint8_t receivedAddress

Matching address send by master. 7-bits plus R/nW bit0

uint32_t eventMask

Mask of enabled events.

uint8_t *rxData

Transfer buffer for receive data

const uint8_t *txData

Transfer buffer for transmit data

size_t txSize

Transfer size

size_t rxSize

Transfer size

size_t transferredCount

Number of bytes transferred during this transfer.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for *kI2C_SlaveCompletionEvent*.

struct *_i2c_slave_handle*

#include <fsl_i2c.h> I2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

volatile *i2c_slave_transfer_t* transfer

I2C slave transfer.

volatile bool isBusy

Whether transfer is busy.

volatile *i2c_slave_fsm_t* slaveFsm

slave transfer state machine.

i2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

2.22 INPUTMUX: Input Multiplexing Driver

FSL_INPUTMUX_DRIVER_VERSION

Group interrupt driver version for SDK.

void INPUTMUX_Init(void *base)

Initialize INPUTMUX peripheral.

This function enables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

void INPUTMUX_AttachSignal(void *base, uint32_t index, inputmux_connection_t connection)

Attaches a signal.

This function attaches multiplexed signals from INPUTMUX to target signals. For example, to attach GPIO PORT0 Pin 5 to PINT peripheral, do the following:

```
INPUTMUX_AttachSignal(INPUTMUX, 2, kINPUTMUX_GpioPort0Pin5ToPintsel);
```

In this example, INTMUX has 8 registers for PINT, PINT_SEL0~PINT_SEL7. With parameter *index* specified as 2, this function configures register PINT_SEL2.

Parameters

- base – Base address of the INPUTMUX peripheral.
- index – The serial number of destination register in the group of INPUTMUX registers with same name.

- connection – Applies signal from source signals collection to target signal.

Return values

None. –

void INPUTMUX_EnableSignal(void *base, inputmux_signal_t signal, bool enable)

Enable/disable a signal.

This function gates the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.
- signal – Enable signal register id and bit offset.
- enable – Selects enable or disable.

Return values

None. –

void INPUTMUX_Deinit(void *base)

Deinitialize INPUTMUX peripheral.

This function disables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

2.23 Common Driver

FSL_COMMON_DRIVER_VERSION

common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE

No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART

Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART

Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI

Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)

Set the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)

Clear the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)

Toggle the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)

For the variable at address *address*, clear the bits specifiled by *clearBits* and set the bits specifiled by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true** , else return **false** .

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_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

`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*.

`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 I2C status codes.

enumerator `kStatusGroup_LPSCI`

Group number for LPSCI status codes.

enumerator `kStatusGroup_LPUART`

Group number for LPUART status codes.

enumerator `kStatusGroup_SPI`

Group number for SPI status code.

enumerator `kStatusGroup_XRDC`

Group number for XRDC status code.

enumerator `kStatusGroup_SEMA42`

Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
Group number for debug console status codes.

enumerator kStatusGroup_SEMC
Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
Starting number for application groups.

enumerator kStatusGroup_IAP
Group number for IAP status codes

enumerator kStatusGroup_SFA
Group number for SFA status codes

enumerator kStatusGroup_SPC
Group number for SPC status codes.

enumerator kStatusGroup_PUF
Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes

enumerator kStatusGroup_VBAT
Group number for VBAT status codes

enumerator kStatusGroup_XSPI
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator `kStatusGroup_LIST`
Group number for List status codes.

enumerator `kStatusGroup_OSA`
Group number for OSA status codes.

enumerator `kStatusGroup_COMMON_TASK`
Group number for Common task status codes.

enumerator `kStatusGroup_MSG`
Group number for messaging status codes.

enumerator `kStatusGroup_SDK_OCOTP`
Group number for OCOTP status codes.

enumerator `kStatusGroup_SDK_FLEXSPINOR`
Group number for FLEXSPINOR status codes.

enumerator `kStatusGroup_CODEEC`
Group number for codec status codes.

enumerator `kStatusGroup_ASRC`
Group number for codec status ASRC.

enumerator `kStatusGroup_OTFAD`
Group number for codec status codes.

enumerator `kStatusGroup_SDIO SLV`
Group number for SDIO SLV 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_QUEUE DSPI`
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.

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_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.24 GPIO: General Purpose I/O

void GPIO_PortInit(GPIO_Type *base, uint32_t port)

Initializes the GPIO peripheral.

This function ungates the GPIO clock.

Parameters

- base – GPIO peripheral base pointer.
- port – GPIO port number.

void GPIO_PinInit(GPIO_Type *base, uint32_t port, 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, either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or 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(Typically GPIO)
- port – GPIO port number

- pin – GPIO pin number
- config – GPIO pin configuration pointer

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t port, uint32_t pin, uint8_t output)

Sets the output level of the one GPIO pin to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- 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 uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t port, uint32_t pin)

Reads the current input value of the GPIO PIN.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

FSL_GPIO_DRIVER_VERSION

LPC GPIO driver version.

enum _gpio_pin_direction

LPC GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput

Set current pin as digital input

enumerator kGPIO_DigitalOutput

Set current pin as digital output

typedef enum _gpio_pin_direction gpio_pin_direction_t

LPC GPIO direction definition.

typedef struct _gpio_pin_config gpio_pin_config_t

The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t port, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number

- mask – GPIO pin number macro

static inline void GPIO_PortClear(GPIO_Type *base, uint32_t port, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- mask – GPIO pin number macro

static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t port, uint32_t mask)

Reverses current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- mask – GPIO pin number macro

struct __gpio_pin_config

#include <fsl_gpio.h> The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

Public Members

gpio_pin_direction_t pinDirection

GPIO direction, input or output

uint8_t outputLogic

Set default output logic, no use in input

2.25 IOCON: I/O pin configuration

FSL_IOCON_DRIVER_VERSION

IOCON driver version.

typedef struct *iocon_group* iocon_group_t

Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format.

__STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t port, uint8_t pin, uint32_t modefunc)

Sets I/O Control pin mux.

Parameters

- base – : The base of IOCON peripheral on the chip
- port – : GPIO port to mux
- pin – : GPIO pin to mux
- modefunc – : OR'ed values of type IOCON_*

Returns

Nothing

```
__STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base,  
const iocon_group_t *pinArray, uint32_t arrayLength)
```

Set all I/O Control pin muxing.

Parameters

- base – : The base of IOCON peripheral on the chip
- pinArray – : Pointer to array of pin mux selections
- arrayLength – : Number of entries in pinArray

Returns

Nothing

FSL_COMPONENT_ID

IOCON_FUNC0

IOCON function and mode selection definitions.

Note: See the User Manual for specific modes and functions supported by the various pins.
Selects pin function 0

IOCON_FUNC1

Selects pin function 1

IOCON_FUNC2

Selects pin function 2

IOCON_FUNC3

Selects pin function 3

IOCON_FUNC4

Selects pin function 4

IOCON_FUNC5

Selects pin function 5

IOCON_FUNC6

Selects pin function 6

IOCON_FUNC7

Selects pin function 7

struct __iocon_group

#include <fsl_iocon.h> Array of IOCON pin definitions passed to IOCON_SetPinMuxing()
must be in this format.

2.26 Mailbox

```
static inline void MAILBOX_Init(MAILBOX_Type *base)
```

Initializes the MAILBOX module.

This function enables the MAILBOX clock only.

Parameters

- base – MAILBOX peripheral base address.

static inline void MAILBOX_Deinit(MAILBOX_Type *base)

De-initializes the MAILBOX module.

This function disables the MAILBOX clock only.

Parameters

- base – MAILBOX peripheral base address.

FSL_MAILBOX_DRIVER_VERSION

MAILBOX driver version.

static inline uint32_t MAILBOX_GetMutex(MAILBOX_Type *base)

Get MUTEX state and lock mutex.

Note: Returns '1' if the mutex was taken or '0' if another resources has the mutex locked. Once a mutex is taken, it can be returned with the MAILBOX_SetMutex() function.

Parameters

- base – MAILBOX peripheral base address.

Returns

See note

static inline void MAILBOX_SetMutex(MAILBOX_Type *base)

Set MUTEX state.

Note: Sets mutex state to '1' and allows other resources to get the mutex.

Parameters

- base – MAILBOX peripheral base address.

FSL_COMPONENT_ID

2.27 MRT: Multi-Rate Timer

void MRT_Init(MRT_Type *base, const *mrt_config_t* *config)

Ungates the MRT clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the MRT driver.

Parameters

- base – Multi-Rate timer peripheral base address
- config – Pointer to user's MRT config structure. If MRT has MULTITASK bit field in MODCFG register, param config is useless.

void MRT_Deinit(MRT_Type *base)

Gate the MRT clock.

Parameters

- base – Multi-Rate timer peripheral base address

static inline void MRT_GetDefaultConfig(*mrt_config_t* *config)

Fill in the MRT config struct with the default settings.

The default values are:

```
config->enableMultiTask = false;
```

Parameters

- config – Pointer to user's MRT config structure.

static inline void MRT_SetupChannelMode(MRT_Type *base, *mrt_chnl_t* channel, const *mrt_timer_mode_t* mode)

Sets up an MRT channel mode.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Channel that is being configured.
- mode – Timer mode to use for the channel.

static inline void MRT_EnableInterrupts(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Enables the MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline void MRT_DisableInterrupts(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Disables the selected MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline uint32_t MRT_GetEnabledInterrupts(MRT_Type *base, *mrt_chnl_t* channel)

Gets the enabled MRT interrupts.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline uint32_t MRT_GetStatusFlags(MRT_Type *base, *mrt_chnl_t* channel)

Gets the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration `mrt_status_flags_t`

```
static inline void MRT_ClearStatusFlags(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Clears the MRT status flags.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `mrt_status_flags_t`

```
void MRT_UpdateTimerPeriod(MRT_Type *base, mrt_chnl_t channel, uint32_t count, bool  
                           immediateLoad)
```

Used to update the timer period in units of count.

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number
- `count` – Timer period in units of ticks
- `immediateLoad` – `true`: Load the new value immediately into the TIMER register; `false`: Load the new value at the end of current timer interval

```
static inline uint32_t MRT_GetCurrentTimerCount(MRT_Type *base, mrt_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 usec or msec

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number

Returns

Current timer counting value in ticks

```
static inline void MRT_StartTimer(MRT_Type *base, mrt_chnl_t channel, uint32_t count)
```

Starts the timer counting.

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.
- `count` – Timer period in units of ticks. Count can contain the LOAD bit, which control the force load feature.

static inline void MRT_StopTimer(MRT_Type *base, *mrt_chnl_t* channel)

Stops the timer counting.

This function stops the timer from counting.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

static inline uint32_t MRT_GetIdleChannel(MRT_Type *base)

Find the available channel.

This function returns the lowest available channel number.

Parameters

- `base` – Multi-Rate timer peripheral base address

static inline void MRT_ReleaseChannel(MRT_Type *base, *mrt_chnl_t* channel)

Release the channel when the timer is using the multi-task mode.

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling `MRT_GetIdleChannel()` for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

FSL_MRT_DRIVER_VERSION

enum __mrt_chnl

List of MRT channels.

Values:

enumerator kMRT_Channel_0

MRT channel number 0

enumerator kMRT_Channel_1

MRT channel number 1

enumerator kMRT_Channel_2

MRT channel number 2

enumerator kMRT_Channel_3

MRT channel number 3

enum __mrt_timer_mode

List of MRT timer modes.

Values:

enumerator kMRT_RepeatMode

Repeat Interrupt mode

enumerator kMRT_OneShotMode

One-shot Interrupt mode

enumerator kMRT_OneShotStallMode

One-shot stall mode

enum __mrt_interrupt_enable

List of MRT interrupts.

Values:

enumerator kMRT_TimerInterruptEnable

Timer interrupt enable

enum __mrt_status_flags

List of MRT status flags.

Values:

enumerator kMRT_TimerInterruptFlag

Timer interrupt flag

enumerator kMRT_TimerRunFlag

Indicates state of the timer

typedef enum __mrt_chnl mrt_chnl_t

List of MRT channels.

typedef enum __mrt_timer_mode mrt_timer_mode_t

List of MRT timer modes.

typedef enum __mrt_interrupt_enable mrt_interrupt_enable_t

List of MRT interrupts.

typedef enum __mrt_status_flags mrt_status_flags_t

List of MRT status flags.

typedef struct __mrt_config mrt_config_t

MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the MRT_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

struct __mrt_config

#include <fsl_mrt.h> MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the MRT_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 enableMultiTask

true: Timers run in multi-task mode; false: Timers run in hardware status mode

2.28 OSTIMER: OS Event Timer Driver

`void OSTIMER_Init(OSTIMER_Type *base)`

Initializes an OSTIMER by turning its bus clock on.

`void OSTIMER_Deinit(OSTIMER_Type *base)`

Deinitializes a OSTIMER instance.

This function shuts down OSTIMER bus clock

Parameters

- `base` – OSTIMER peripheral base address.

`uint64_t OSTIMER_GrayToDecimal(uint64_t gray)`

Translate the value from gray-code to decimal.

Parameters

- `gray` – The gray value input.

Returns

The decimal value.

`static inline uint64_t OSTIMER_DecimalToGray(uint64_t dec)`

Translate the value from decimal to gray-code.

Parameters

- `dec` – The decimal value.

Returns

The gray code of the input value.

`uint32_t OSTIMER_GetStatusFlags(OSTIMER_Type *base)`

Get OSTIMER status Flags.

This returns the status flag. Currently, only match interrupt flag can be got.

Parameters

- `base` – OSTIMER peripheral base address.

Returns

status register value

`void OSTIMER_ClearStatusFlags(OSTIMER_Type *base, uint32_t mask)`

Clear Status Interrupt Flags.

This clears intrrupt status flag. Currently, only match interrupt flag can be cleared.

Parameters

- `base` – OSTIMER peripheral base address.
- `mask` – Clear bit mask.

Returns

none

`status_t OSTIMER_SetMatchRawValue(OSTIMER_Type *base, uint64_t count, ostimer_callback_t cb)`

Set the match raw value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central EVTIMER. Please note that, the data format may be gray-code, if so, please using `OSTIMER_SetMatchValue()`.

Parameters

- `base` – OSTIMER peripheral base address.
- `count` – OSTIMER timer match value.(Value may be gray-code format)
- `cb` – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

- `kStatus__Success` – - Set match raw value and enable interrupt Successfully.
- `kStatus__Fail` – - Set match raw value fail.

`status_t` OSTIMER_SetMatchValue(OSTIMER_Type *base, uint64_t count, *ostimer_callback_t* cb)

Set the match value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central OS TIMER.

Parameters

- `base` – OSTIMER peripheral base address.
- `count` – OSTIMER timer match value.(Value is decimal format, and this value will be translate to Gray code in API if the IP counter is gray encoded.)
- `cb` – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

- `kStatus__Success` – - Set match value and enable interrupt Successfully.
- `kStatus__Fail` – - Set match value fail.

static inline void OSTIMER_SetMatchRegister(OSTIMER_Type *base, uint64_t value)

Set value to OSTIMER MATCH register directly.

This function writes the input value to OSTIMER MATCH register directly, it does not touch any other registers. Note that, the data format is gray-code. The function OSTIMER_DecimalToGray could convert decimal value to gray code.

Parameters

- `base` – OSTIMER peripheral base address.
- `value` – OSTIMER timer match value (Value is gray-code format).

static inline uint64_t OSTIMER_GetMatchRegister(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get the match value from OSTIMER. The value of timer match is gray code format.

Parameters

- `base` – OSTIMER peripheral base address.

Returns

Value of match register, data format is gray code.

static inline uint64_t OSTIMER_GetMatchValue(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get a match value from OSTIMER.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of match register.

```
static inline void OSTIMER_EnableMatchInterrupt(OSTIMER_Type *base)
```

Enable the OSTIMER counter match interrupt.

Enable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- base – OSTIMER peripheral base address.

```
static inline void OSTIMER_DisableMatchInterrupt(OSTIMER_Type *base)
```

Disable the OSTIMER counter match interrupt.

Disable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- base – OSTIMER peripheral base address.

```
static inline uint64_t OSTIMER_GetCurrentTimerRawValue(OSTIMER_Type *base)
```

Get current timer raw count value from OSTIMER.

This function will get the timer count value from OS timer register. The raw value of timer count may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of OSTIMER, may be gray code format.

```
uint64_t OSTIMER_GetCurrentValue(OSTIMER_Type *base)
```

Get current timer count value from OSTIMER.

This function will get a decimal timer count value. If the RAW value of timer count is gray code format, it will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of OSTIMER which will be formatted to decimal value.

```
static inline uint64_t OSTIMER_GetCaptureRawValue(OSTIMER_Type *base)
```

Get the capture value from OSTIMER.

This function will get a captured value from OSTIMER. The Raw value of timer capture may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of capture register, data format may be gray code.

```
uint64_t OSTIMER_GetCaptureValue(OSTIMER_Type *base)
```

Get the capture value from OSTIMER.

This function will get a capture decimal-value from OSTIMER. If the RAW value of timer count is gray code format, it will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of capture register, data format is decimal.

void OSTIMER_HandleIRQ(OSTIMER_Type *base, *ostimer_callback_t* cb)

OS timer interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in OSTIMER_SetMatchValue()). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- base – OS timer peripheral base address.
- cb – callback scheduled for this instance of OS timer

Returns

none

FSL_OSTIMER_DRIVER_VERSION

OSTIMER driver version.

enum *_ostimer_flags*

OSTIMER status flags.

Values:

enumerator kOSTIMER_MatchInterruptFlag

Match interrupt flag bit, sets if the match value was reached.

typedef void (**ostimer_callback_t*)(void)

ostimer callback function.

2.29 PINT: Pin Interrupt and Pattern Match Driver

FSL_PINT_DRIVER_VERSION

enum *_pint_pin_enable*

PINT Pin Interrupt enable type.

Values:

enumerator kPINT_PinIntEnableNone

Do not generate Pin Interrupt

enumerator kPINT_PinIntEnableRiseEdge

Generate Pin Interrupt on rising edge

enumerator kPINT_PinIntEnableFallEdge

Generate Pin Interrupt on falling edge

enumerator kPINT_PinIntEnableBothEdges

Generate Pin Interrupt on both edges

enumerator kPINT_PinIntEnableLowLevel

Generate Pin Interrupt on low level

enumerator kPINT_PinIntEnableHighLevel

Generate Pin Interrupt on high level

enum `_pint_int`

PINT Pin Interrupt type.

Values:

enumerator `kPINT_PinInt0`

Pin Interrupt 0

enumerator `kPINT_SecPinInt0`

Secure Pin Interrupt 0

enum `_pint_pmatch_input_src`

PINT Pattern Match bit slice input source type.

Values:

enumerator `kPINT_PatternMatchInp0Src`

Input source 0

enumerator `kPINT_PatternMatchInp1Src`

Input source 1

enumerator `kPINT_PatternMatchInp2Src`

Input source 2

enumerator `kPINT_PatternMatchInp3Src`

Input source 3

enumerator `kPINT_PatternMatchInp4Src`

Input source 4

enumerator `kPINT_PatternMatchInp5Src`

Input source 5

enumerator `kPINT_PatternMatchInp6Src`

Input source 6

enumerator `kPINT_PatternMatchInp7Src`

Input source 7

enumerator `kPINT_SecPatternMatchInp0Src`

Input source 0

enumerator `kPINT_SecPatternMatchInp1Src`

Input source 1

enum `_pint_pmatch_bslice`

PINT Pattern Match bit slice type.

Values:

enumerator `kPINT_PatternMatchBSlice0`

Bit slice 0

enumerator `kPINT_SecPatternMatchBSlice0`

Bit slice 0

enum `_pint_pmatch_bslice_cfg`

PINT Pattern Match configuration type.

Values:

enumerator `kPINT_PatternMatchAlways`

Always Contributes to product term match

```

enumerator kPINT_PatternMatchStickyRise
    Sticky Rising edge
enumerator kPINT_PatternMatchStickyFall
    Sticky Falling edge
enumerator kPINT_PatternMatchStickyBothEdges
    Sticky Rising or Falling edge
enumerator kPINT_PatternMatchHigh
    High level
enumerator kPINT_PatternMatchLow
    Low level
enumerator kPINT_PatternMatchNever
    Never contributes to product term match
enumerator kPINT_PatternMatchBothEdges
    Either rising or falling edge
typedef enum _pint_pin_enable pint_pin_enable_t
    PINT Pin Interrupt enable type.
typedef enum _pint_int pint_pin_int_t
    PINT Pin Interrupt type.
typedef enum _pint_pmatch_input_src pint_pmatch_input_src_t
    PINT Pattern Match bit slice input source type.
typedef enum _pint_pmatch_bslic pint_pmatch_bslic_t
    PINT Pattern Match bit slice type.
typedef enum _pint_pmatch_bslic_cfg pint_pmatch_bslic_cfg_t
    PINT Pattern Match configuration type.
typedef struct _pint_status pint_status_t
    PINT event status.
typedef void (*pint_cb_t)(pint_pin_int_t pintr, pint_status_t *status)
    PINT Callback function.
typedef struct _pint_pmatch_cfg pint_pmatch_cfg_t
void PINT_Init(PINT_Type *base)
    Initialize PINT peripheral.
    This function initializes the PINT peripheral and enables the clock.

```

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```

void PINT_SetCallback(PINT_Type *base, pint_cb_t callback)
    Set PINT callback.

```

This function set the callback for PINT interrupt handler.

Parameters

- base – Base address of the PINT peripheral.
- callback – Callback.

Return values

None. –

```
void PINT_PinInterruptConfig(PINT_Type *base, pint_pin_int_t intr, pint_pin_enable_t enable)
```

Configure PINT peripheral pin interrupt.

This function configures a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- intr – Pin interrupt.
- enable – Selects detection logic.

Return values

None. –

```
void PINT_PinInterruptGetConfig(PINT_Type *base, pint_pin_int_t pintr, pint_pin_enable_t *enable)
```

Get PINT peripheral pin interrupt configuration.

This function returns the configuration of a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.
- enable – Pointer to store the detection logic.

Return values

None. –

```
void PINT_PinInterruptClrStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt status only when the pin was triggered by edge-sensitive.

This function clears the selected pin interrupt status.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Get Selected pin interrupt status.

This function returns the selected pin interrupt status.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

status – = 0 No pin interrupt request. = 1 Selected Pin interrupt request active.

```
void PINT_PinInterruptClrStatusAll(PINT_Type *base)
```

Clear all pin interrupts status only when pins were triggered by edge-sensitive.

This function clears the status of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatusAll(PINT_Type *base)
```

Get all pin interrupts status.

This function returns the status of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the status of corresponding pin interrupt.
= 0 No pin interrupt request. = 1 Pin interrupt request active.

```
static inline void PINT_PinInterruptClrFallFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt fall flag.

This function clears the selected pin interrupt fall flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Get selected pin interrupt fall flag.

This function returns the selected pin interrupt fall flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

flag – = 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrFallFlagAll(PINT_Type *base)
```

Clear all pin interrupt fall flags.

This function clears the fall flag for all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlagAll(PINT_Type *base)
```

Get all pin interrupt fall flags.

This function returns the fall flag of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

flags – Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt rise flag.

This function clears the selected pin interrupt rise flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Get selected pin interrupt rise flag.

This function returns the selected pin interrupt rise flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

flag – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlagAll(PINT_Type *base)
```

Clear all pin interrupt rise flags.

This function clears the rise flag for all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlagAll(PINT_Type *base)
```

Get all pin interrupt rise flags.

This function returns the rise flag of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

flags – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
void PINT_PatternMatchConfig(PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
```

Configure PINT pattern match.

This function configures a given pattern match bit slice.

Parameters

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.
- cfg – Pointer to bit slice configuration.

Return values

None. –


```
void PINT_PatternMatchGetConfig(PINT_Type *base, pint_pmatch_bslice_t bslice,  
                                pint_pmatch_cfg_t *cfg)
```

Get PINT pattern match configuration.

This function returns the configuration of a given pattern match bit slice.

Parameters

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.
- cfg – Pointer to bit slice configuration.

Return values

None. –

```
static inline uint32_t PINT_PatternMatchGetStatus(PINT_Type *base, pint_pmatch_bslice_t  
                                                  bslice)
```

Get pattern match bit slice status.

This function returns the status of selected bit slice.

Parameters

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.

Return values

status – = 0 Match has not been detected. = 1 Match has been detected.

```
static inline uint32_t PINT_PatternMatchGetStatusAll(PINT_Type *base)
```

Get status of all pattern match bit slices.

This function returns the status of all bit slices.

Parameters

- base – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the match status of corresponding bit slice.
= 0 Match has not been detected. = 1 Match has been detected.

```
uint32_t PINT_PatternMatchResetDetectLogic(PINT_Type *base)
```

Reset pattern match detection logic.

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

- base – Base address of the PINT peripheral.

Return values

pmstatus – Each bit position indicates the match status of corresponding bit slice.
= 0 Match was detected. = 1 Match was not detected.

```
static inline void PINT_PatternMatchEnable(PINT_Type *base)
```

Enable pattern match function.

This function enables the pattern match function.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchDisable(PINT_Type *base)

Disable pattern match function.

This function disables the pattern match function.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchEnableRXEV(PINT_Type *base)

Enable RXEV output.

This function enables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchDisableRXEV(PINT_Type *base)

Disable RXEV output.

This function disables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_EnableCallback(PINT_Type *base)

Enable callback.

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_DisableCallback(PINT_Type *base)

Disable callback.

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

- base – Base address of the peripheral.

Return values

None. –

void PINT_Deinit(PINT_Type *base)

Deinitialize PINT peripheral.

This function disables the PINT clock.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

`void PINT_EnableCallbackByIndex(PINT_Type *base, pint_pin_int_t pintIdx)`
enable callback by pin index.

This function enables callback by pin index instead of enabling all pins.

Parameters

- base – Base address of the peripheral.
- pintIdx – pin index.

Return values

None. –

`void PINT_DisableCallbackByIndex(PINT_Type *base, pint_pin_int_t pintIdx)`
disable callback by pin index.

This function disables callback by pin index instead of disabling all pins.

Parameters

- base – Base address of the peripheral.
- pintIdx – pin index.

Return values

None. –

`PINT_USE_LEGACY_CALLBACK`

`PININT_BITSLICE_SRC_START`

`PININT_BITSLICE_SRC_MASK`

`PININT_BITSLICE_CFG_START`

`PININT_BITSLICE_CFG_MASK`

`PININT_BITSLICE_ENDP_MASK`

`PINT_PIN_INT_LEVEL`

`PINT_PIN_INT_EDGE`

`PINT_PIN_INT_FALL_OR_HIGH_LEVEL`

`PINT_PIN_INT_RISE`

`PINT_PIN_RISE_EDGE`

`PINT_PIN_FALL_EDGE`

`PINT_PIN_BOTH_EDGE`

`PINT_PIN_LOW_LEVEL`

`PINT_PIN_HIGH_LEVEL`

`struct __pint_status`

#include <fsl_pint.h> PINT event status.

`struct __pint_pmatch_cfg`

#include <fsl_pint.h>

2.30 PLU: Programmable Logic Unit

void PLU_Init(PLU_Type *base)

Enable the PLU clock and reset the module.

Note: This API should be called at the beginning of the application using the PLU driver.

Parameters

- base – PLU peripheral base address

void PLU_Deinit(PLU_Type *base)

Gate the PLU clock.

Parameters

- base – PLU peripheral base address

static inline void PLU_SetLutInputSource(PLU_Type *base, *plu_lut_index_t* lutIndex,
plu_lut_in_index_t lutInIndex, *plu_lut_input_source_t*
inputSrc)

Set Input source of LUT.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs. For each LUT, the slot associated with the output from LUTn itself is tied low.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see *plu_lut_index_t* typedef enumeration).
- lutInIndex – LUT input index (see *plu_lut_in_index_t* typedef enumeration).
- inputSrc – LUT input source (see *plu_lut_input_source_t* typedef enumeration).

static inline void PLU_SetOutputSource(PLU_Type *base, *plu_output_index_t* outputIndex,
plu_output_source_t outputSrc)

Set Output source of PLU.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs.

Parameters

- base – PLU peripheral base address.
- outputIndex – PLU output index (see *plu_output_index_t* typedef enumeration).
- outputSrc – PLU output source (see *plu_output_source_t* typedef enumeration).

static inline void PLU_SetLutTruthTable(PLU_Type *base, *plu_lut_index_t* lutIndex, uint32_t
truthTable)

Set Truth Table of LUT.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see *plu_lut_index_t* typedef enumeration).
- truthTable – Truth Table value.

static inline uint32_t PLU_ReadOutputState(PLU_Type *base)

Read the current state of the 8 designated PLU Outputs.

Note: The PLU bus clock must be re-enabled prior to reading the Output Register if PLU bus clock is shut-off.

Parameters

- base – PLU peripheral base address.

Returns

Current PLU output state value.

void PLU_GetDefaultWakeIntConfig(*plu_wakeint_config_t* *config)

Gets an available pre-defined settings for wakeup/interrupt control.

This function initializes the initial configuration structure with an available settings. The default values are:

```
config->filterMode = kPLU_WAKEINT_FILTER_MODE_BYPASS;
config->clockSource = kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC;
```

Parameters

- config – Pointer to configuration structure.

void PLU_EnableWakeIntRequest(PLU_Type *base, uint32_t interruptMask, const *plu_wakeint_config_t* *config)

Enable PLU outputs wakeup/interrupt request.

This function enables Any of the eight selected PLU outputs to contribute to an asynchronous wake-up or an interrupt request.

Note: If a PLU_CLKIN is provided, the raw wake-up/interrupt request will be set on the rising-edge of the PLU_CLKIN whenever the raw request signal is high. This registered signal will be glitch-free and just use the default wakeint config by PLU_GetDefaultWakeIntConfig(). If not, have to specify the filter mode and clock source to eliminate the glitches caused by long and widely disparate delays through the network of LUTs making up the PLU. This way may increase power consumption in low-power operating modes and inject delay before the wake-up/interrupt request is generated.

Parameters

- base – PLU peripheral base address.
- interruptMask – PLU interrupt mask (see *_plu_interrupt_mask* enumeration).
- config – Pointer to configuration structure (see *plu_wakeint_config_t* typedef enumeration)

static inline void PLU_LatchInterrupt(PLU_Type *base)

Latch an interrupt.

This function latches the interrupt and then it can be cleared with PLU_ClearLatchedInterrupt().

Note: This mode is not compatible with use of the glitch filter. If this bit is set, the FILTER MODE should be set to kPLU_WAKEINT_FILTER_MODE_BYPASS (Bypass Mode) and PLU_CLKIN should be provided. If this bit is set, the wake-up/interrupt request will be set on the rising-edge of PLU_CLKIN whenever the raw wake-up/interrupt signal is high. The request must be cleared by software.

Parameters

- base – PLU peripheral base address.

void PLU_ClearLatchedInterrupt(PLU_Type *base)

Clear the latched interrupt.

This function clears the wake-up/interrupt request flag latched by PLU_LatchInterrupt()

Note: It is not necessary for the PLU bus clock to be enabled in order to write-to or read-back this bit.

Parameters

- base – PLU peripheral base address.

FSL_PLU_DRIVER_VERSION

Version 2.2.1

enum __plu_lut_index

Index of LUT.

Values:

enumerator kPLU_LUT_0

5-input Look-up Table 0

enumerator kPLU_LUT_1

5-input Look-up Table 1

enumerator kPLU_LUT_2

5-input Look-up Table 2

enumerator kPLU_LUT_3

5-input Look-up Table 3

enumerator kPLU_LUT_4

5-input Look-up Table 4

enumerator kPLU_LUT_5

5-input Look-up Table 5

enumerator kPLU_LUT_6

5-input Look-up Table 6

enumerator kPLU_LUT_7

5-input Look-up Table 7

enumerator kPLU_LUT_8

5-input Look-up Table 8

enumerator kPLU_LUT_9

5-input Look-up Table 9

enumerator kPLU_LUT_10

5-input Look-up Table 10

enumerator kPLU_LUT_11

5-input Look-up Table 11

enumerator kPLU_LUT_12

5-input Look-up Table 12

enumerator kPLU_LUT_13

5-input Look-up Table 13

enumerator kPLU_LUT_14

5-input Look-up Table 14

enumerator kPLU_LUT_15
5-input Look-up Table 15

enumerator kPLU_LUT_16
5-input Look-up Table 16

enumerator kPLU_LUT_17
5-input Look-up Table 17

enumerator kPLU_LUT_18
5-input Look-up Table 18

enumerator kPLU_LUT_19
5-input Look-up Table 19

enumerator kPLU_LUT_20
5-input Look-up Table 20

enumerator kPLU_LUT_21
5-input Look-up Table 21

enumerator kPLU_LUT_22
5-input Look-up Table 22

enumerator kPLU_LUT_23
5-input Look-up Table 23

enumerator kPLU_LUT_24
5-input Look-up Table 24

enumerator kPLU_LUT_25
5-input Look-up Table 25

enum _plu_lut_in_index

Inputs of LUT. 5 input present for each LUT.

Values:

enumerator kPLU_LUT_IN_0
LUT input 0

enumerator kPLU_LUT_IN_1
LUT input 1

enumerator kPLU_LUT_IN_2
LUT input 2

enumerator kPLU_LUT_IN_3
LUT input 3

enumerator kPLU_LUT_IN_4
LUT input 4

enum _plu_lut_input_source

Available sources of LUT input.

Values:

enumerator kPLU_LUT_IN_SRC_PLU_IN_0
Select PLU input 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_1
Select PLU input 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_2
Select PLU input 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_3
Select PLU input 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_4
Select PLU input 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_5
Select PLU input 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_0
Select LUT output 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_1
Select LUT output 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_2
Select LUT output 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_3
Select LUT output 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_4
Select LUT output 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_5
Select LUT output 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_6
Select LUT output 6 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_7
Select LUT output 7 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_8
Select LUT output 8 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_9
Select LUT output 9 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_10
Select LUT output 10 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_11
Select LUT output 11 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_12
Select LUT output 12 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_13
Select LUT output 13 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_14
Select LUT output 14 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_15
Select LUT output 15 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_16
Select LUT output 16 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_17
 Select LUT output 17 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_18
 Select LUT output 18 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_19
 Select LUT output 19 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_20
 Select LUT output 20 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_21
 Select LUT output 21 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_22
 Select LUT output 22 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_23
 Select LUT output 23 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_24
 Select LUT output 24 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_25
 Select LUT output 25 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_0
 Select Flip-Flops state 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_1
 Select Flip-Flops state 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_2
 Select Flip-Flops state 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_3
 Select Flip-Flops state 3 to be connected to LUTn Input x

enum _plu_output_index

PLU output multiplexer registers.

Values:

enumerator kPLU_OUTPUT_0
 PLU OUTPUT 0

enumerator kPLU_OUTPUT_1
 PLU OUTPUT 1

enumerator kPLU_OUTPUT_2
 PLU OUTPUT 2

enumerator kPLU_OUTPUT_3
 PLU OUTPUT 3

enumerator kPLU_OUTPUT_4
 PLU OUTPUT 4

enumerator kPLU_OUTPUT_5
 PLU OUTPUT 5

enumerator kPLU_OUTPUT_6

PLU OUTPUT 6

enumerator kPLU_OUTPUT_7

PLU OUTPUT 7

enum __plu_output_source

Available sources of PLU output.

Values:

enumerator kPLU_OUT_SRC_LUT_0

Select LUT0 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_1

Select LUT1 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_2

Select LUT2 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_3

Select LUT3 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_4

Select LUT4 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_5

Select LUT5 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_6

Select LUT6 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_7

Select LUT7 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_8

Select LUT8 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_9

Select LUT9 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_10

Select LUT10 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_11

Select LUT11 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_12

Select LUT12 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_13

Select LUT13 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_14

Select LUT14 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_15

Select LUT15 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_16

Select LUT16 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_17

Select LUT17 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_18

Select LUT18 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_19

Select LUT19 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_20

Select LUT20 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_21

Select LUT21 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_22

Select LUT22 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_23

Select LUT23 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_24

Select LUT24 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_25

Select LUT25 output to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_0

Select Flip-Flops state(0) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_1

Select Flip-Flops state(1) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_2

Select Flip-Flops state(2) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_3

Select Flip-Flops state(3) to be connected to PLU output

enum _plu_interrupt_mask

The enumerator of PLU Interrupt.

Values:

enumerator kPLU_OUTPUT_0_INTERRUPT_MASK

Select PLU output 0 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_1_INTERRUPT_MASK

Select PLU output 1 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_2_INTERRUPT_MASK

Select PLU output 2 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_3_INTERRUPT_MASK

Select PLU output 3 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_4_INTERRUPT_MASK

Select PLU output 4 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_5_INTERRUPT_MASK

Select PLU output 5 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_6_INTERRUPT_MASK

Select PLU output 6 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_7_INTERRUPT_MASK

Select PLU output 7 contribute to interrupt/wake-up generation

enum __plu_wakeint_filter_mode

Control input of the PLU, add filtering for glitch.

Values:

enumerator kPLU_WAKEINT_FILTER_MODE_BYPASS

Select Bypass mode

enumerator kPLU_WAKEINT_FILTER_MODE_1_CLK_PERIOD

Filter 1 clock period

enumerator kPLU_WAKEINT_FILTER_MODE_2_CLK_PERIOD

Filter 2 clock period

enumerator kPLU_WAKEINT_FILTER_MODE_3_CLK_PERIOD

Filter 3 clock period

enum __plu_wakeint_filter_clock_source

Clock source for filter mode.

Values:

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC

Select the 1MHz low-power oscillator as the filter clock

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_12MHZ_FRO

Select the 12MHz FRO as the filter clock

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_ALT

Select a third clock source

typedef enum __plu_lut_index plu_lut_index_t

Index of LUT.

typedef enum __plu_lut_in_index plu_lut_in_index_t

Inputs of LUT. 5 input present for each LUT.

typedef enum __plu_lut_input_source plu_lut_input_source_t

Available sources of LUT input.

typedef enum __plu_output_index plu_output_index_t

PLU output multiplexer registers.

typedef enum __plu_output_source plu_output_source_t

Available sources of PLU output.

typedef enum __plu_wakeint_filter_mode plu_wakeint_filter_mode_t

Control input of the PLU, add filtering for glitch.

typedef enum __plu_wakeint_filter_clock_source plu_wakeint_filter_clock_source_t

Clock source for filter mode.

typedef struct __plu_wakeint_config plu_wakeint_config_t

Wake configuration.

struct __plu_wakeint_config

#include <fsl_plu.h> Wake configuration.

Public Members

plu_wakeint_filter_mode_t filterMode

Filter Mode.

plu_wakeint_filter_clock_source_t clockSource

The clock source for filter mode.

2.31 PUF: Physical Unclonable Function

FSL_PUF_DRIVER_VERSION

PUF driver version. Version 2.2.0.

Current version: 2.2.0

Change log:

- 2.0.0
 - Initial version.
- 2.0.1
 - Fixed puf_wait_usec function optimization issue.
- 2.0.2
 - Add PUF configuration structure and support for PUF SRAM controller. Remove magic constants.
- 2.0.3
 - Fix MISRA C-2012 issue.
- 2.1.0
 - Align driver with PUF SRAM controller registers on LPCXpresso55s16.
 - Update initialization logic .
- 2.1.1
 - Fix ARMGCC build warning .
- 2.1.2
 - Update: Add automatic big to little endian swap for user (pre-shared) keys destined to secret hardware bus (PUF key index 0).
- 2.1.3
 - Fix MISRA C-2012 issue.
- 2.1.4
 - Replace register uint32_t ticksCount with volatile uint32_t ticksCount in puf_wait_usec() to prevent optimization out delay loop.
- 2.1.5
 - Use common SDK delay in puf_wait_usec()
- 2.1.6
 - Changed wait time in PUF_Init(), when initialization fails it will try PUF_Powercycle() with shorter time. If this shorter time will also fail, initialization will be tried with worst case time as before.
- 2.2.0

- Add support for kPUF_KeySlot4.
- Add new PUF_ClearKey() function, that clears a desired PUF internal HW key register.

enum __puf_key_index_register

Values:

enumerator kPUF_KeyIndex_00

enumerator kPUF_KeyIndex_01

enumerator kPUF_KeyIndex_02

enumerator kPUF_KeyIndex_03

enumerator kPUF_KeyIndex_04

enumerator kPUF_KeyIndex_05

enumerator kPUF_KeyIndex_06

enumerator kPUF_KeyIndex_07

enumerator kPUF_KeyIndex_08

enumerator kPUF_KeyIndex_09

enumerator kPUF_KeyIndex_10

enumerator kPUF_KeyIndex_11

enumerator kPUF_KeyIndex_12

enumerator kPUF_KeyIndex_13

enumerator kPUF_KeyIndex_14

enumerator kPUF_KeyIndex_15

enum __puf_min_max

Values:

enumerator kPUF_KeySizeMin

enumerator kPUF_KeySizeMax

enumerator kPUF_KeyIndexMax

enum __puf_key_slot

PUF key slot.

Values:

enumerator kPUF_KeySlot0

PUF key slot 0

enumerator kPUF_KeySlot1

PUF key slot 1

PUF status return codes.

Values:

enumerator kStatus_EnrollNotAllowed

```
enumerator kStatus_StartNotAllowed

typedef enum _puf_key_index_register puf_key_index_register_t

typedef enum _puf_min_max puf_min_max_t

typedef enum _puf_key_slot puf_key_slot_t
    PUF key slot.

PUF_GET_KEY_CODE_SIZE_FOR_KEY_SIZE(x)
    Get Key Code size in bytes from key size in bytes at compile time.

PUF_MIN_KEY_CODE_SIZE

PUF_ACTIVATION_CODE_SIZE

KEystore_PUF_DISCHARGE_TIME_FIRST_TRY_MS

KEystore_PUF_DISCHARGE_TIME_MAX_MS

struct puf_config_t
    #include <fsl_puf.h>
```

2.32 RTC: Real Time Clock

```
void RTC_Init(RTC_Type *base)
    Un-gate the RTC clock and enable the RTC oscillator.
```

Note: This API should be called at the beginning of the application using the RTC driver.

Parameters

- base – RTC peripheral base address

```
static inline void RTC_Deinit(RTC_Type *base)
    Stop the timer and gate the RTC clock.
```

Parameters

- base – RTC peripheral base address

```
status_t RTC_SetDatetime(RTC_Type *base, const rtc_datetime_t *datetime)
    Set the RTC date and time according to the given time structure.
```

The RTC counter must be stopped prior to calling this function as writes to the RTC seconds register will fail if the RTC counter is running.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to structure where the date and time details to set are stored

Returns

kStatus_Success: Success in setting the time and starting the RTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

```
void RTC_GetDatetime(RTC_Type *base, rtc_datetime_t *datetime)
```

Get the RTC time and stores it in the given time structure.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to structure where the date and time details are stored.

```
status_t RTC_SetAlarm(RTC_Type *base, const rtc_datetime_t *alarmTime)
```

Set 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 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)
```

Return the RTC alarm time.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to structure where the alarm date and time details are stored.

```
static inline void RTC_EnableWakeupTimer(RTC_Type *base, bool enable)
```

Enable the RTC wake-up timer (1KHZ).

After calling this function, the RTC driver will use/un-use the RTC wake-up (1KHZ) at the same time.

Parameters

- base – RTC peripheral base address
- enable – Use/Un-use the RTC wake-up timer.
 - true: Use RTC wake-up timer at the same time.
 - false: Un-use RTC wake-up timer, RTC only use the normal seconds timer by default.

```
static inline uint32_t RTC_GetEnabledWakeupTimer(RTC_Type *base)
```

Get the enabled status of the RTC wake-up timer (1KHZ).

Parameters

- base – RTC peripheral base address

Returns

The enabled status of RTC wake-up timer (1KHZ).

```
static inline void RTC_EnableSubsecCounter(RTC_Type *base, bool enable)
```

Enable the RTC Sub-second counter (32KHZ).

Note: Only enable sub-second counter after RTC_ENA bit has been set to 1.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC sub-second counter.
 - true: Enable RTC sub-second counter.
 - false: Disable RTC sub-second counter.

static inline uint32_t RTC_GetSubsecValue(const RTC_Type *base)

A read of 32KHZ sub-seconds counter.

Parameters

- base – RTC peripheral base address

Returns

Current value of the SUBSEC register

static inline void RTC_EnableWakeUpTimerInterruptFromDPD(RTC_Type *base, bool enable)

Enable the wake-up timer interrupt from deep power down mode.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable wake-up timer interrupt from deep power down mode.
 - true: Enable wake-up timer interrupt from deep power down mode.
 - false: Disable wake-up timer interrupt from deep power down mode.

static inline void RTC_EnableAlarmTimerInterruptFromDPD(RTC_Type *base, bool enable)

Enable the alarm timer interrupt from deep power down mode.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable alarm timer interrupt from deep power down mode.
 - true: Enable alarm timer interrupt from deep power down mode.
 - false: Disable alarm timer interrupt from deep power down mode.

static inline void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superseded by RTC_EnableAlarmTimerInterruptFromDPD and RTC_EnableWakeUpTimerInterruptFromDPD

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

static inline void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superceded by `RTC_EnableAlarmTimerInterruptFromDPD` and `RTC_EnableWakeUpTimerInterruptFromDPD`

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`

```
static inline uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)
```

Get the enabled RTC interrupts.

Deprecated:

Do not use this function. It will be deleted in next release version.

Parameters

- `base` – RTC peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `rtc_interrupt_enable_t`

```
static inline uint32_t RTC_GetStatusFlags(RTC_Type *base)
```

Get 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`

```
static inline void RTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)
```

Clear 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_EnableTimer(RTC_Type *base, bool enable)
```

Enable the RTC timer counter.

After calling this function, the RTC inner counter increments once a second when only using the RTC seconds timer (1hz), while the RTC inner wake-up timer countdown once a millisecond when using RTC wake-up timer (1KHZ) at the same time. RTC timer contain two timers, one is the RTC normal seconds timer, the other one is the RTC wake-up timer, the RTC enable bit is the master switch for the whole RTC timer, so user can use the RTC seconds (1HZ) timer independly, but they can't use the RTC wake-up timer (1KHZ) independently.

Parameters

- `base` – RTC peripheral base address
- `enable` – Enable/Disable RTC Timer counter.
 - `true`: Enable RTC Timer counter.
 - `false`: Disable RTC Timer counter.

```
static inline void RTC_StartTimer(RTC_Type *base)
```

Starts the RTC time counter.

Deprecated:

Do not use this function. It has been superceded by RTC_EnableTimer

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.

Deprecated:

Do not use this function. It has been superceded by RTC_EnableTimer

RTC's seconds register can be written to only when the timer is stopped.

Parameters

- base – RTC peripheral base address

FSL_RTC_DRIVER_VERSION

Version 2.2.0

```
enum _rtc_interrupt_enable
```

List of RTC interrupts.

Values:

enumerator kRTC_AlarmInterruptEnable

Alarm interrupt.

enumerator kRTC_WakeupInterruptEnable

Wake-up interrupt.

```
enum _rtc_status_flags
```

List of RTC flags.

Values:

enumerator kRTC_AlarmFlag

Alarm flag

enumerator kRTC_WakeupFlag

1kHz wake-up timer flag

```
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 struct _rtc_datetime rtc_datetime_t
```

Structure is used to hold the date and time.

```
static inline void RTC__SetSecondsTimerMatch(RTC_Type *base, uint32_t matchValue)
```

Set the RTC seconds timer (1HZ) MATCH value.

Parameters

- base – RTC peripheral base address
- matchValue – The value to be set into the RTC MATCH register

```
static inline uint32_t RTC__GetSecondsTimerMatch(RTC_Type *base)
```

Read actual RTC seconds timer (1HZ) MATCH value.

Parameters

- base – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) MATCH value.

```
static inline void RTC__SetSecondsTimerCount(RTC_Type *base, uint32_t countValue)
```

Set the RTC seconds timer (1HZ) COUNT value.

Parameters

- base – RTC peripheral base address
- countValue – The value to be loaded into the RTC COUNT register

```
static inline uint32_t RTC__GetSecondsTimerCount(RTC_Type *base)
```

Read the actual RTC seconds timer (1HZ) COUNT value.

Parameters

- base – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) COUNT value.

```
static inline void RTC__SetWakeupCount(RTC_Type *base, uint16_t wakeupValue)
```

Enable the RTC wake-up timer (1KHZ) and set countdown value to the RTC WAKE register.

Parameters

- base – RTC peripheral base address
- wakeupValue – The value to be loaded into the WAKE register in RTC wake-up timer (1KHZ).

```
static inline uint16_t RTC__GetWakeupCount(RTC_Type *base)
```

Read the actual value from the WAKE register value in RTC wake-up timer (1KHZ)

Read the WAKE register twice and compare the result, if the value match, the time can be used.

Parameters

- base – RTC peripheral base address

Returns

The actual value of the WAKE register value in RTC wake-up timer (1KHZ).

```
static inline void RTC__Reset(RTC_Type *base)
```

Perform a software reset on the RTC module.

This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.

Parameters

- base – RTC peripheral base address

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.

2.33 SCTimer: SCTimer/PWM (SCT)

status_t SCTIMER_Init(SCT_Type *base, const sctimer_config_t *config)

Ungates the SCTimer clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the SCTimer driver.

Parameters

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

Returns

kStatus_Success indicates success; Else indicates failure.

void SCTIMER_Deinit(SCT_Type *base)

Gates the SCTimer clock.

Parameters

- base – SCTimer peripheral base address

void SCTIMER_GetDefaultConfig(sctimer_config_t *config)

Fills in the SCTimer configuration structure with the default settings.

The default values are:

```
config->enableCounterUnify = true;
config->clockMode = kSCTIMER_System_ClockMode;
config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
config->enableBidirection_l = false;
config->enableBidirection_h = false;
config->prescale_l = 0U;
```

(continues on next page)

(continued from previous page)

```
config->prescale_h = 0U;  
config->outInitState = 0U;  
config->inputsync = 0xFU;
```

Parameters

- config – Pointer to the user configuration structure.

```
status_t SCTIMER_SetupPwm(SCT_Type *base, const sctimer_pwm_signal_param_t  
                          *pwmParams, sctimer_pwm_mode_t mode, uint32_t  
                          pwmFreq_Hz, uint32_t srcClock_Hz, uint32_t *event)
```

Configures the PWM signal parameters.

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function `SCTIMER_GetCurrentStateNumber()`. The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

Note: When setting PWM output from multiple output pins, they all should use the same PWM mode i.e all PWM's should be either edge-aligned or center-aligned. When using this API, the PWM signal frequency of all the initialized channels must be the same. Otherwise all the initialized channels' PWM signal frequency is equal to the last call to the API's `pwmFreq_Hz`.

Parameters

- base – SCTimer peripheral base address
- pwmParams – PWM parameters to configure the output
- mode – PWM operation mode, options available in enumeration `sctimer_pwm_mode_t`
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – SCTimer counter clock in Hz
- event – Pointer to a variable where the PWM period event number is stored

Returns

`kStatus_Success` on success `kStatus_Fail` If we have hit the limit in terms of number of events created or if an incorrect PWM duty cycle is passed in.

```
void SCTIMER_UpdatePwmDutycycle(SCT_Type *base, sctimer_out_t output, uint8_t  
                                dutyCyclePercent, uint32_t event)
```

Updates the duty cycle of an active PWM signal.

Before calling this function, the counter is set to operate as one 32-bit counter (unify bit is set to 1).

Parameters

- base – SCTimer peripheral base address
- output – The output to configure
- dutyCyclePercent – New PWM pulse width; the value should be between 1 to 100

- `event` – Event number associated with this PWM signal. This was returned to the user by the function `SCTIMER_SetupPwm()`.

`static inline void SCTIMER_EnableInterrupts(SCT_Type *base, uint32_t mask)`

Enables the selected SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `sctimer_interrupt_enable_t`

`static inline void SCTIMER_DisableInterrupts(SCT_Type *base, uint32_t mask)`

Disables the selected SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `sctimer_interrupt_enable_t`

`static inline uint32_t SCTIMER_GetEnabledInterrupts(SCT_Type *base)`

Gets the enabled SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `sctimer_interrupt_enable_t`

`static inline uint32_t SCTIMER_GetStatusFlags(SCT_Type *base)`

Gets the SCTimer status flags.

Parameters

- `base` – SCTimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `sctimer_status_flags_t`

`static inline void SCTIMER_ClearStatusFlags(SCT_Type *base, uint32_t mask)`

Clears the SCTimer status flags.

Parameters

- `base` – SCTimer peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `sctimer_status_flags_t`

`static inline void SCTIMER_StartTimer(SCT_Type *base, uint32_t countertoStart)`

Starts the SCTimer counter.

Note: In 16-bit mode, we can enable both Counter_L and Counter_H, In 32-bit mode, we only can select Counter_U.

Parameters

- `base` – SCTimer peripheral base address
- `countertoStart` – The SCTimer counters to enable. This is a logical OR of members of the enumeration `sctimer_counter_t`.

```
static inline void SCTIMER_StopTimer(SCT_Type *base, uint32_t countertoStop)
```

Halts the SCTimer counter.

Parameters

- base – SCTimer peripheral base address
- countertoStop – The SCTimer counters to stop. This is a logical OR of members of the enumeration `sctimer_counter_t`.

```
status_t SCTIMER_CreateAndScheduleEvent(SCT_Type *base, sctimer_event_t howToMonitor,  
                                         uint32_t matchValue, uint32_t whichIO,  
                                         sctimer_counter_t whichCounter, uint32_t *event)
```

Create an event that is triggered on a match or IO and schedule in current state.

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

Parameters

- base – SCTimer peripheral base address
- howToMonitor – Event type; options are available in the enumeration `sctimer_interrupt_enable_t`
- matchValue – The match value that will be programmed to a match register
- whichIO – The input or output that will be involved in event triggering. This field is ignored if the event type is “match only”
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Pointer to a variable where the new event number is stored

Returns

`kStatus_Success` on success `kStatus_Error` if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

```
void SCTIMER_ScheduleEvent(SCT_Type *base, uint32_t event)
```

Enable an event in the current state.

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function `SCTIMER_SetupPwm()` or function `SCTIMER_CreateAndScheduleEvent()`.

Parameters

- base – SCTimer peripheral base address
- event – Event number to enable in the current state

```
status_t SCTIMER_IncreaseState(SCT_Type *base)
```

Increase the state by 1.

All future events created by calling the function `SCTIMER_ScheduleEvent()` will be enabled in this new state.

Parameters

- base – SCTimer peripheral base address

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of states used

uint32_t SCTIMER_GetCurrentState(SCT_Type *base)

Provides the current state.

User can use this to set the next state by calling the function SCTIMER_SetupNextStateAction().

Parameters

- base – SCTimer peripheral base address

Returns

The current state

static inline void SCTIMER_SetCounterState(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t state)

Set the counter current state.

The function is to set the state variable bit field of STATE register. Writing to the STATE_L, STATE_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- state – The counter current state number (only support range from 0~31).

static inline uint16_t SCTIMER_GetCounterState(SCT_Type *base, *sctimer_counter_t* whichCounter)

Get the counter current state value.

The function is to get the state variable bit field of STATE register.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The the counter current state value.

status_t SCTIMER_SetupCaptureAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t *captureRegister, uint32_t event)

Setup capture of the counter value on trigger of a selected event.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- captureRegister – Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
- event – Event number that will trigger the capture

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

void SCTIMER_SetCallback(SCT_Type *base, *sctimer_event_callback_t* callback, uint32_t event)

Receive notification when the event trigger an interrupt.

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

Parameters

- base – SCTimer peripheral base address
- event – Event number that will trigger the interrupt
- callback – Function to invoke when the event is triggered

static inline void SCTIMER_SetupStateLdMethodAction(SCT_Type *base, uint32_t event, bool fgLoad)

Change the load method of transition to the specified state.

Change the load method of transition, it will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- event – Event number that will change the method to trigger the state transition
- fgLoad – The method to load highest-numbered event occurring for that state to the STATE register.
 - true: Load the STATEV value to STATE when the event occurs to be the next state.
 - false: Add the STATEV value to STATE when the event occurs to be the next state.

static inline void SCTIMER_SetupNextStateActionwithLdMethod(SCT_Type *base, uint32_t nextState, uint32_t event, bool fgLoad)

Transition to the specified state with Load method.

This transition will be triggered by the event number that is passed in by the user, the method decide how to load the highest-numbered event occurring for that state to the STATE register.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition
- fgLoad – The method to load the highest-numbered event occurring for that state to the STATE register.
 - true: Load the STATEV value to STATE when the event occurs to be the next state.
 - false: Add the STATEV value to STATE when the event occurs to be the next state.

```
static inline void SCTIMER_SetupNextStateAction(SCT_Type *base, uint32_t nextState, uint32_t event)
```

Transition to the specified state.

Deprecated:

Do not use this function. It has been superceded by SCTIMER_SetupNextStateActionwithLdMethod

This transition will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition

```
static inline void SCTIMER_SetupEventActiveDirection(SCT_Type *base,
                                                    sctimer_event_active_direction_t
                                                    activeDirection, uint32_t event)
```

Setup event active direction when the counters are operating in BIDIR mode.

Parameters

- base – SCTimer peripheral base address
- activeDirection – Event generation active direction, see sctimer_event_active_direction_t.
- event – Event number that need setup the active direction.

```
static inline void SCTIMER_SetupOutputSetAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Set the Output.

This output will be set when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to set
- event – Event number that will trigger the output change

```
static inline void SCTIMER_SetupOutputClearAction(SCT_Type *base, uint32_t whichIO,
                                                  uint32_t event)
```

Clear the Output.

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to clear
- event – Event number that will trigger the output change

```
void SCTIMER_SetupOutputToggleAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Toggle the output level.

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

- `base` – SCTimer peripheral base address
- `whichIO` – The output to toggle
- `event` – Event number that will trigger the output change

```
static inline void SCTIMER_SetupCounterLimitAction(SCT_Type *base, sctimer_counter_t  
                                                  whichCounter, uint32_t event)
```

Limit the running counter.

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Event number that will trigger the counter to be limited

```
static inline void SCTIMER_SetupCounterStopAction(SCT_Type *base, sctimer_counter_t  
                                                  whichCounter, uint32_t event)
```

Stop the running counter.

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Event number that will trigger the counter to be stopped

```
static inline void SCTIMER_SetupCounterStartAction(SCT_Type *base, sctimer_counter_t  
                                                  whichCounter, uint32_t event)
```

Re-start the stopped counter.

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Event number that will trigger the counter to re-start

```
static inline void SCTIMER_SetupCounterHaltAction(SCT_Type *base, sctimer_counter_t  
                                                  whichCounter, uint32_t event)
```

Halt the running counter.

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the `SCTIMER_StartTimer()` function.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Event number that will trigger the counter to be halted

```
static inline void SCTIMER_SetupDmaTriggerAction(SCT_Type *base, uint32_t dmaNumber,  
                                                uint32_t event)
```

Generate a DMA request.

DMA request will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- dmaNumber – The DMA request to generate
- event – Event number that will trigger the DMA request

```
static inline void SCTIMER_SetCOUNTValue(SCT_Type *base, sctimer_counter_t whichCounter,  
                                          uint32_t value)
```

Set the value of counter.

The function is to set the value of Count register, Writing to the COUNT_L, COUNT_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- value – the counter value update to the COUNT register.

```
static inline uint32_t SCTIMER_GetCOUNTValue(SCT_Type *base, sctimer_counter_t  
                                              whichCounter)
```

Get the value of counter.

The function is to read the value of Count register, software can read the counter registers at any time..

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The value of counter selected.

```
static inline void SCTIMER_SetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Set the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be set.
- state – The state value in which the event is enabled to occur.

```
static inline void SCTIMER_ClearEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Clear the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be clear.
- state – The state value in which the event is disabled to occur.

static inline bool SCTIMER_GetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
Get the state mask bit field of EV_STATE register.

Note: This function is to check whether the event is enabled in a specific state.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be read.
- state – The state value.

Returns

The the state mask bit field of EV_STATE register.

- true: The event is enable in state.
- false: The event is disable in state.

static inline uint32_t SCTIMER_GetCaptureValue(SCT_Type *base, *sctimer_counter_t*
whichCounter, uint8_t capChannel)

Get the value of capture register.

This function returns the captured value upon occurrence of the events selected by the corresponding Capture Control registers occurred.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- capChannel – SCTimer capture register of capture channel.

Returns

The SCTimer counter value at which this register was last captured.

void SCTIMER_EventHandleIRQ(SCT_Type *base)

SCTimer interrupt handler.

Parameters

- base – SCTimer peripheral base address.

FSL_SCTIMER_DRIVER_VERSION

Version

enum _sctimer_pwm_mode

SCTimer PWM operation modes.

Values:

enumerator kSCTIMER_EdgeAlignedPwm

Edge-aligned PWM

enumerator kSCTIMER_CenterAlignedPwm

Center-aligned PWM

enum _sctimer_counter

SCTimer counters type.

Values:

enumerator kSCTIMER_Counter_L
16-bit Low counter.

enumerator kSCTIMER_Counter_H
16-bit High counter.

enumerator kSCTIMER_Counter_U
32-bit Unified counter.

enum _sctimer_input

List of SCTimer input pins.

Values:

enumerator kSCTIMER_Input_0
SCTIMER input 0

enumerator kSCTIMER_Input_1
SCTIMER input 1

enumerator kSCTIMER_Input_2
SCTIMER input 2

enumerator kSCTIMER_Input_3
SCTIMER input 3

enumerator kSCTIMER_Input_4
SCTIMER input 4

enumerator kSCTIMER_Input_5
SCTIMER input 5

enumerator kSCTIMER_Input_6
SCTIMER input 6

enumerator kSCTIMER_Input_7
SCTIMER input 7

enum _sctimer_out

List of SCTimer output pins.

Values:

enumerator kSCTIMER_Out_0
SCTIMER output 0

enumerator kSCTIMER_Out_1
SCTIMER output 1

enumerator kSCTIMER_Out_2
SCTIMER output 2

enumerator kSCTIMER_Out_3
SCTIMER output 3

enumerator kSCTIMER_Out_4
SCTIMER output 4

enumerator kSCTIMER_Out_5
SCTIMER output 5

enumerator kSCTIMER_Out_6
SCTIMER output 6

enumerator kSCTIMER_Out_7

SCTIMER output 7

enumerator kSCTIMER_Out_8

SCTIMER output 8

enumerator kSCTIMER_Out_9

SCTIMER output 9

enum __sctimer_pwm_level_select

SCTimer PWM output pulse mode: high-true, low-true or no output.

Values:

enumerator kSCTIMER_LowTrue

Low true pulses

enumerator kSCTIMER_HighTrue

High true pulses

enum __sctimer_clock_mode

SCTimer clock mode options.

Values:

enumerator kSCTIMER_System_ClockMode

System Clock Mode

enumerator kSCTIMER_Sampled_ClockMode

Sampled System Clock Mode

enumerator kSCTIMER_Input_ClockMode

SCT Input Clock Mode

enumerator kSCTIMER_Asynchronous_ClockMode

Asynchronous Mode

enum __sctimer_clock_select

SCTimer clock select options.

Values:

enumerator kSCTIMER_Clock_On_Rise_Input_0

Rising edges on input 0

enumerator kSCTIMER_Clock_On_Fall_Input_0

Falling edges on input 0

enumerator kSCTIMER_Clock_On_Rise_Input_1

Rising edges on input 1

enumerator kSCTIMER_Clock_On_Fall_Input_1

Falling edges on input 1

enumerator kSCTIMER_Clock_On_Rise_Input_2

Rising edges on input 2

enumerator kSCTIMER_Clock_On_Fall_Input_2

Falling edges on input 2

enumerator kSCTIMER_Clock_On_Rise_Input_3

Rising edges on input 3

enumerator kSCTIMER_Clock_On_Fall_Input_3

Falling edges on input 3

enumerator kSCTIMER_Clock_On_Rise_Input_4

Rising edges on input 4

enumerator kSCTIMER_Clock_On_Fall_Input_4

Falling edges on input 4

enumerator kSCTIMER_Clock_On_Rise_Input_5

Rising edges on input 5

enumerator kSCTIMER_Clock_On_Fall_Input_5

Falling edges on input 5

enumerator kSCTIMER_Clock_On_Rise_Input_6

Rising edges on input 6

enumerator kSCTIMER_Clock_On_Fall_Input_6

Falling edges on input 6

enumerator kSCTIMER_Clock_On_Rise_Input_7

Rising edges on input 7

enumerator kSCTIMER_Clock_On_Fall_Input_7

Falling edges on input 7

enum _sctimer_conflict_resolution

SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Values:

enumerator kSCTIMER_ResolveNone

No change

enumerator kSCTIMER_ResolveSet

Set output

enumerator kSCTIMER_ResolveClear

Clear output

enumerator kSCTIMER_ResolveToggle

Toggle output

enum _sctimer_event_active_direction

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

Values:

enumerator kSCTIMER_ActiveIndependent

This event is triggered regardless of the count direction.

enumerator kSCTIMER_ActiveInCountUp

This event is triggered only during up-counting when BIDIR = 1.

enumerator kSCTIMER_ActiveInCountDown

This event is triggered only during down-counting when BIDIR = 1.

enum _sctimer_event

List of SCTimer event types.

Values:

enumerator kSCTIMER_InputLowOrMatchEvent
enumerator kSCTIMER_InputRiseOrMatchEvent
enumerator kSCTIMER_InputFallOrMatchEvent
enumerator kSCTIMER_InputHighOrMatchEvent
enumerator kSCTIMER_MatchEventOnly
enumerator kSCTIMER_InputLowEvent
enumerator kSCTIMER_InputRiseEvent
enumerator kSCTIMER_InputFallEvent
enumerator kSCTIMER_InputHighEvent
enumerator kSCTIMER_InputLowAndMatchEvent
enumerator kSCTIMER_InputRiseAndMatchEvent
enumerator kSCTIMER_InputFallAndMatchEvent
enumerator kSCTIMER_InputHighAndMatchEvent
enumerator kSCTIMER_OutputLowOrMatchEvent
enumerator kSCTIMER_OutputRiseOrMatchEvent
enumerator kSCTIMER_OutputFallOrMatchEvent
enumerator kSCTIMER_OutputHighOrMatchEvent
enumerator kSCTIMER_OutputLowEvent
enumerator kSCTIMER_OutputRiseEvent
enumerator kSCTIMER_OutputFallEvent
enumerator kSCTIMER_OutputHighEvent
enumerator kSCTIMER_OutputLowAndMatchEvent
enumerator kSCTIMER_OutputRiseAndMatchEvent
enumerator kSCTIMER_OutputFallAndMatchEvent
enumerator kSCTIMER_OutputHighAndMatchEvent

enum _sctimer_interrupt_enable

List of SCTimer interrupts.

Values:

enumerator kSCTIMER_Event0InterruptEnable
 Event 0 interrupt
enumerator kSCTIMER_Event1InterruptEnable
 Event 1 interrupt

enumerator kSCTIMER_Event2InterruptEnable

Event 2 interrupt

enumerator kSCTIMER_Event3InterruptEnable

Event 3 interrupt

enumerator kSCTIMER_Event4InterruptEnable

Event 4 interrupt

enumerator kSCTIMER_Event5InterruptEnable

Event 5 interrupt

enumerator kSCTIMER_Event6InterruptEnable

Event 6 interrupt

enumerator kSCTIMER_Event7InterruptEnable

Event 7 interrupt

enumerator kSCTIMER_Event8InterruptEnable

Event 8 interrupt

enumerator kSCTIMER_Event9InterruptEnable

Event 9 interrupt

enumerator kSCTIMER_Event10InterruptEnable

Event 10 interrupt

enumerator kSCTIMER_Event11InterruptEnable

Event 11 interrupt

enumerator kSCTIMER_Event12InterruptEnable

Event 12 interrupt

enum _sctimer_status_flags

List of SCTimer flags.

Values:

enumerator kSCTIMER_Event0Flag

Event 0 Flag

enumerator kSCTIMER_Event1Flag

Event 1 Flag

enumerator kSCTIMER_Event2Flag

Event 2 Flag

enumerator kSCTIMER_Event3Flag

Event 3 Flag

enumerator kSCTIMER_Event4Flag

Event 4 Flag

enumerator kSCTIMER_Event5Flag

Event 5 Flag

enumerator kSCTIMER_Event6Flag

Event 6 Flag

enumerator kSCTIMER_Event7Flag

Event 7 Flag

enumerator `kSCTIMER_Event8Flag`
Event 8 Flag

enumerator `kSCTIMER_Event9Flag`
Event 9 Flag

enumerator `kSCTIMER_Event10Flag`
Event 10 Flag

enumerator `kSCTIMER_Event11Flag`
Event 11 Flag

enumerator `kSCTIMER_Event12Flag`
Event 12 Flag

enumerator `kSCTIMER_BusErrorLFlag`
Bus error due to write when L counter was not halted

enumerator `kSCTIMER_BusErrorHFlag`
Bus error due to write when H counter was not halted

typedef enum `_sctimer_pwm_mode` `sctimer_pwm_mode_t`
SCTimer PWM operation modes.

typedef enum `_sctimer_counter` `sctimer_counter_t`
SCTimer counters type.

typedef enum `_sctimer_input` `sctimer_input_t`
List of SCTimer input pins.

typedef enum `_sctimer_out` `sctimer_out_t`
List of SCTimer output pins.

typedef enum `_sctimer_pwm_level_select` `sctimer_pwm_level_select_t`
SCTimer PWM output pulse mode: high-true, low-true or no output.

typedef struct `_sctimer_pwm_signal_param` `sctimer_pwm_signal_param_t`
Options to configure a SCTimer PWM signal.

typedef enum `_sctimer_clock_mode` `sctimer_clock_mode_t`
SCTimer clock mode options.

typedef enum `_sctimer_clock_select` `sctimer_clock_select_t`
SCTimer clock select options.

typedef enum `_sctimer_conflict_resolution` `sctimer_conflict_resolution_t`
SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

typedef enum `_sctimer_event_active_direction` `sctimer_event_active_direction_t`
List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

typedef enum `_sctimer_event` `sctimer_event_t`
List of SCTimer event types.

typedef void (*`sctimer_event_callback_t`)(void)
SCTimer callback typedef.

typedef enum `_sctimer_interrupt_enable` `sctimer_interrupt_enable_t`
List of SCTimer interrupts.

```
typedef enum _sctimer_status_flags sctimer_status_flags_t
```

List of SCTimer flags.

```
typedef struct _sctimer_config sctimer_config_t
```

SCTimer configuration structure.

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the SCTMR_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

```
SCT_EV_STATE_STATEMSK(x)
```

```
struct _sctimer_pwm_signal_param
```

#include <fsl_sctimer.h> Options to configure a SCTimer PWM signal.

Public Members

```
sctimer_out_t output
```

The output pin to use to generate the PWM signal

```
sctimer_pwm_level_select_t level
```

PWM output active level select.

```
uint8_t dutyCyclePercent
```

PWM pulse width, value should be between 0 to 100 0 = always inactive signal (0% duty cycle) 100 = always active signal (100% duty cycle).

```
struct _sctimer_config
```

#include <fsl_sctimer.h> SCTimer configuration structure.

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the SCTMR_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 enableCounterUnify
```

true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters. User can use the 16-bit low counter and the 16-bit high counters at the same time; for Hardware limit, user can not use unified 32-bit counter and any 16-bit low/high counter at the same time.

```
sctimer_clock_mode_t clockMode
```

SCT clock mode value

```
sctimer_clock_select_t clockSelect
```

SCT clock select value

```
bool enableBidirection_l
```

true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter

```
bool enableBidirection_h
```

true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter. This field is used only if the enableCounterUnify is set to false

uint8_t prescale_l

Prescale value to produce the L or unified counter clock

uint8_t prescale_h

Prescale value to produce the H counter clock. This field is used only if the enable-CounterUnify is set to false

uint8_t outInitState

Defines the initial output value

uint8_t inputsync

SCT INSYNC value, INSYNC field in the CONFIG register, from bit9 to bit 16. it is used to define synchronization for input N: bit 9 = input 0 bit 10 = input 1 bit 11 = input 2 bit 12 = input 3 All other bits are reserved (bit13 ~bit 16). How User to set the the value for the member inputsync. IE: delay for input0, and input 1, bypasses for input 2 and input 3 MACRO definition in user level. #define INPUTSYNC0 (0U) #define INPUTSYNC1 (1U) #define INPUTSYNC2 (2U) #define INPUTSYNC3 (3U) User Code. sctimerInfo.inputsync = (1 « INPUTSYNC2) | (1 « INPUTSYNC3);

2.34 SPI: Serial Peripheral Interface Driver

2.35 SPI DMA Driver

```
status_t SPI_MasterTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                           spi_dma_callback_t callback, void *userData,
                                           dma_handle_t *txHandle, dma_handle_t
                                           *rxHandle)
```

Initialize the SPI master DMA handle.

This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
status_t SPI_MasterTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t
                               *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

```
status_t SPI_MasterHalfDuplexTransferDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                         spi_half_duplex_transfer_t *xfer)
```

Transfers a block of data using a DMA method.

This function using polling way to do the first half transimission and using DMA way to do the srcond half transimission, the transfer mechanism is half-duplex. When do the second half transimission, code will return right away. When all data is transferred, the callback function is called.

Parameters

- base – SPI base pointer
- handle – A pointer to the spi_master_dma_handle_t structure which stores the transfer state.
- xfer – A pointer to the spi_half_duplex_transfer_t structure.

Returns

status of status_t.

```
static inline status_t SPI_SlaveTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t
                                                       *handle, spi_dma_callback_t callback,
                                                       void *userData, dma_handle_t
                                                       *txHandle, dma_handle_t *rxHandle)
```

Initialize the SPI slave DMA handle.

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
static inline status_t SPI_SlaveTransferDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                           spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

void SPI_MasterTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

status_t SPI_MasterTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t *handle, size_t *count)

Gets the master DMA transfer remaining bytes.

This function gets the master DMA transfer remaining bytes.

Parameters

- base – SPI peripheral base address.
- handle – A pointer to the spi_dma_handle_t structure which stores the transfer state.
- count – A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

static inline void SPI_SlaveTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

static inline status_t SPI_SlaveTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t *handle, size_t *count)

Gets the slave DMA transfer remaining bytes.

This function gets the slave DMA transfer remaining bytes.

Parameters

- base – SPI peripheral base address.
- handle – A pointer to the spi_dma_handle_t structure which stores the transfer state.
- count – A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

FSL_SPI_DMA_DRIVER_VERSION

SPI DMA driver version 2.1.1.

typedef struct *_spi_dma_handle* spi_dma_handle_t

typedef void (*spi_dma_callback_t)(SPI_Type *base, *spi_dma_handle_t* *handle, *status_t* status, void *userData)

SPI DMA callback called at the end of transfer.

struct *_spi_dma_handle*

#include <fsl_spi_dma.h> SPI DMA transfer handle, users should not touch the content of the handle.

Public Members

volatile bool txInProgress

Send transfer finished

volatile bool rxInProgress

Receive transfer finished

uint8_t bytesPerFrame

Bytes in a frame for SPI transfer

uint8_t lastwordBytes

The Bytes of lastword for master

dma_handle_t *txHandle

DMA handler for SPI send

dma_handle_t *rxHandle

DMA handler for SPI receive

spi_dma_callback_t callback

Callback for SPI DMA transfer

void *userData

User Data for SPI DMA callback

uint32_t state

Internal state of SPI DMA transfer

size_t transferSize

Bytes need to be transfer

uint32_t instance

Index of SPI instance

const uint8_t *txNextData

The pointer of next time tx data

const uint8_t *txEndData

The pointer of end of data

uint8_t *rxNextData

The pointer of next time rx data

uint8_t *rxEndData

The pointer of end of rx data

uint32_t dataBytesEveryTime

Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

2.36 SPI Driver

FSL_SPI_DRIVER_VERSION

SPI driver version.

enum _spi_xfer_option

SPI transfer option.

Values:

enumerator kSPI_FrameDelay

A delay may be inserted, defined in the DLY register.

enumerator kSPI_FrameAssert

SSEL will be deasserted at the end of a transfer

enum _spi_shift_direction

SPI data shifter direction options.

Values:

enumerator kSPI_MsbFirst

Data transfers start with most significant bit.

enumerator kSPI_LsbFirst

Data transfers start with least significant bit.

enum _spi_clock_polarity

SPI clock polarity configuration.

Values:

enumerator kSPI_ClockPolarityActiveHigh

Active-high SPI clock (idles low).

enumerator kSPI_ClockPolarityActiveLow

Active-low SPI clock (idles high).

enum _spi_clock_phase

SPI clock phase configuration.

Values:

enumerator kSPI_ClockPhaseFirstEdge

First edge on SCK occurs at the middle of the first cycle of a data transfer.

enumerator kSPI_ClockPhaseSecondEdge

First edge on SCK occurs at the start of the first cycle of a data transfer.

enum _spi_txfifo_watermark

txFIFO watermark values

Values:

enumerator kSPI_TxFifo0

SPI tx watermark is empty

enumerator kSPI_TxFifo1

SPI tx watermark at 1 item

enumerator kSPI_TxFifo2

SPI tx watermark at 2 items

enumerator kSPI_TxFifo3
SPI tx watermark at 3 items

enumerator kSPI_TxFifo4
SPI tx watermark at 4 items

enumerator kSPI_TxFifo5
SPI tx watermark at 5 items

enumerator kSPI_TxFifo6
SPI tx watermark at 6 items

enumerator kSPI_TxFifo7
SPI tx watermark at 7 items

enum _spi_rxfifo_watermark
rxFIFO watermark values

Values:

enumerator kSPI_RxFifo1
SPI rx watermark at 1 item

enumerator kSPI_RxFifo2
SPI rx watermark at 2 items

enumerator kSPI_RxFifo3
SPI rx watermark at 3 items

enumerator kSPI_RxFifo4
SPI rx watermark at 4 items

enumerator kSPI_RxFifo5
SPI rx watermark at 5 items

enumerator kSPI_RxFifo6
SPI rx watermark at 6 items

enumerator kSPI_RxFifo7
SPI rx watermark at 7 items

enumerator kSPI_RxFifo8
SPI rx watermark at 8 items

enum _spi_data_width
Transfer data width.

Values:

enumerator kSPI_Data4Bits
4 bits data width

enumerator kSPI_Data5Bits
5 bits data width

enumerator kSPI_Data6Bits
6 bits data width

enumerator kSPI_Data7Bits
7 bits data width

enumerator kSPI_Data8Bits
8 bits data width

enumerator kSPI_Data9Bits
9 bits data width

enumerator kSPI_Data10Bits
10 bits data width

enumerator kSPI_Data11Bits
11 bits data width

enumerator kSPI_Data12Bits
12 bits data width

enumerator kSPI_Data13Bits
13 bits data width

enumerator kSPI_Data14Bits
14 bits data width

enumerator kSPI_Data15Bits
15 bits data width

enumerator kSPI_Data16Bits
16 bits data width

enum _spi_ssel
Slave select.

Values:

enumerator kSPI_Ssel0
Slave select 0

enumerator kSPI_Ssel1
Slave select 1

enumerator kSPI_Ssel2
Slave select 2

enumerator kSPI_Ssel3
Slave select 3

enum _spi_spol
ssel polarity

Values:

enumerator kSPI_Spol0ActiveHigh

enumerator kSPI_Spol1ActiveHigh

enumerator kSPI_Spol3ActiveHigh

enumerator kSPI_SpolActiveAllHigh

enumerator kSPI_SpolActiveAllLow

SPI transfer status.

Values:

enumerator kStatus_SPI_Busy
SPI bus is busy

enumerator kStatus_SPI_Idle
SPI is idle

enumerator kStatus_SPI_Error
SPI error

enumerator kStatus_SPI_BaudrateNotSupport
Baudrate is not support in current clock source

enumerator kStatus_SPI_Timeout
SPI timeout polling status flags.

enum __spi_interrupt_enable
SPI interrupt sources.

Values:

enumerator kSPI_RxLvlIrq
Rx level interrupt

enumerator kSPI_TxLvlIrq
Tx level interrupt

enum __spi_statusflags
SPI status flags.

Values:

enumerator kSPI_TxEmptyFlag
txFifo is empty

enumerator kSPI_TxNotFullFlag
txFifo is not full

enumerator kSPI_RxNotEmptyFlag
rxFIFO is not empty

enumerator kSPI_RxFullFlag
rxFIFO is full

typedef enum __spi_xfer_option spi_xfer_option_t
SPI transfer option.

typedef enum __spi_shift_direction spi_shift_direction_t
SPI data shifter direction options.

typedef enum __spi_clock_polarity spi_clock_polarity_t
SPI clock polarity configuration.

typedef enum __spi_clock_phase spi_clock_phase_t
SPI clock phase configuration.

typedef enum __spi_txfifo_watermark spi_txfifo_watermark_t
txFIFO watermark values

typedef enum __spi_rxfifo_watermark spi_rxfifo_watermark_t
rxFIFO watermark values

typedef enum __spi_data_width spi_data_width_t
Transfer data width.

typedef enum __spi_ssel spi_ssel_t
Slave select.

```
typedef enum _spi_spol spi_spol_t  
    ssel polarity
```

```
typedef struct _spi_delay_config spi_delay_config_t  
    SPI delay time configure structure. Note: The DLY register controls several programmable  
    delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The  
    maxinum value of these delay time is 15.
```

```
typedef struct _spi_master_config spi_master_config_t  
    SPI master user configure structure.
```

```
typedef struct _spi_slave_config spi_slave_config_t  
    SPI slave user configure structure.
```

```
typedef struct _spi_transfer spi_transfer_t  
    SPI transfer structure.
```

```
typedef struct _spi_half_duplex_transfer spi_half_duplex_transfer_t  
    SPI half-duplex(master only) transfer structure.
```

```
typedef struct _spi_config spi_config_t  
    Internal configuration structure used in 'spi' and 'spi_dma' driver.
```

```
typedef struct _spi_master_handle spi_master_handle_t  
    Master handle type.
```

```
typedef spi_master_handle_t spi_slave_handle_t  
    Slave handle type.
```

```
typedef void (*spi_master_callback_t)(SPI_Type *base, spi_master_handle_t *handle, status_t  
status, void *userData)  
    SPI master callback for finished transmit.
```

```
typedef void (*spi_slave_callback_t)(SPI_Type *base, spi_slave_handle_t *handle, status_t status,  
void *userData)  
    SPI slave callback for finished transmit.
```

```
typedef void (*flexcomm_spi_master_irq_handler_t)(SPI_Type *base, spi_master_handle_t  
*handle)  
    Typedef for master interrupt handler.
```

```
typedef void (*flexcomm_spi_slave_irq_handler_t)(SPI_Type *base, spi_slave_handle_t *handle)  
    Typedef for slave interrupt handler.
```

```
volatile uint8_t s_dummyData[]  
    SPI default SSEL COUNT.  
    Global variable for dummy data value setting.
```

```
SPI_DUMMYDATA  
    SPI dummy transfer data, the data is sent while txBuff is NULL.
```

```
SPI_RETRY_TIMES  
    Retry times for waiting flag.
```

```
SPI_DATA(n)
```

```
SPI_CTRLMASK
```

```
SPI_ASSERTNUM_SSEL(n)
```

```
SPI_DEASSERTNUM_SSEL(n)
```

SPI_DEASSERT_ALL

SPI_FIFOWR_FLAGS_MASK

SPI_FIFOTRIG_TXLVL_GET(base)

SPI_FIFOTRIG_RXLVL_GET(base)

struct __spi_delay_config

#include <fsl_spi.h> SPI delay time configure structure. Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maximum value of these delay time is 15.

Public Members

uint8_t preDelay

Delay between SSEL assertion and the beginning of transfer.

uint8_t postDelay

Delay between the end of transfer and SSEL deassertion.

uint8_t frameDelay

Delay between frame to frame.

uint8_t transferDelay

Delay between transfer to transfer.

struct __spi_master_config

#include <fsl_spi.h> SPI master user configure structure.

Public Members

bool enableLoopback

Enable loopback for test purpose

bool enableMaster

Enable SPI at initialization time

spi_clock_polarity_t polarity

Clock polarity

spi_clock_phase_t phase

Clock phase

spi_shift_direction_t direction

MSB or LSB

uint32_t baudRate_Bps

Baud Rate for SPI in Hz

spi_data_width_t dataWidth

Width of the data

spi_ssel_t sselNum

Slave select number

spi_spol_t sselPol

Configure active CS polarity

```
uint8_t txWatermark
    txFIFO watermark
uint8_t rxWatermark
    rxFIFO watermark
spi_delay_config_t delayConfig
    Delay configuration.
struct __spi_slave_config
    #include <fsl_spi.h> SPI slave user configure structure.
```

Public Members

```
bool enableSlave
    Enable SPI at initialization time
spi_clock_polarity_t polarity
    Clock polarity
spi_clock_phase_t phase
    Clock phase
spi_shift_direction_t direction
    MSB or LSB
spi_data_width_t dataWidth
    Width of the data
spi_spol_t sselPol
    Configure active CS polarity
uint8_t txWatermark
    txFIFO watermark
uint8_t rxWatermark
    rxFIFO watermark
struct __spi_transfer
    #include <fsl_spi.h> SPI transfer structure.
```

Public Members

```
const uint8_t *txData
    Send buffer
uint8_t *rxData
    Receive buffer
uint32_t configFlags
    Additional option to control transfer, spi_xfer_option_t.
size_t dataSize
    Transfer bytes
struct __spi_half_duplex_transfer
    #include <fsl_spi.h> SPI half-duplex(master only) transfer structure.
```


Public Members

const uint8_t *txData

Send buffer

uint8_t *rxData

Receive buffer

size_t txDataSize

Transfer bytes for transmit

size_t rxDataSize

Transfer bytes

uint32_t configFlags

Transfer configuration flags, spi_xfer_option_t.

bool isPcsAssertInTransfer

If PCS pin keep assert between transmit and receive. true for assert and false for de-assert.

bool isTransmitFirst

True for transmit first and false for receive first.

struct _spi_config

#include <fsl_spi.h> Internal configuration structure used in 'spi' and 'spi_dma' driver.

struct _spi_master_handle

#include <fsl_spi.h> SPI transfer handle structure.

Public Members

const uint8_t *volatile txData

Transfer buffer

uint8_t *volatile rxData

Receive buffer

volatile size_t txRemainingBytes

Number of data to be transmitted [in bytes]

volatile size_t rxRemainingBytes

Number of data to be received [in bytes]

volatile int8_t toReceiveCount

The number of data expected to receive in data width. Since the received count and sent count should be the same to complete the transfer, if the sent count is x and the received count is y, toReceiveCount is x-y.

size_t totalByteCount

A number of transfer bytes

volatile uint32_t state

SPI internal state

spi_master_callback_t callback

SPI callback

void *userData

Callback parameter

uint8_t dataWidth

Width of the data [Valid values: 1 to 16]

uint8_t sselNum

Slave select number to be asserted when transferring data [Valid values: 0 to 3]

uint32_t configFlags

Additional option to control transfer

uint8_t txWatermark

txFIFO watermark

uint8_t rxWatermark

rxFIFO watermark

2.37 SPIFI: SPIFI flash interface driver

```
void SPIFI_TransferTxCreateHandleDMA(SPIFI_Type *base, spifi_dma_handle_t *handle,  
                                     spifi_dma_callback_t callback, void *userData,  
                                     dma_handle_t *dmaHandle)
```

Initializes the SPIFI handle for send which is used in transactional functions and set the callback.

Parameters

- base – SPIFI peripheral base address
- handle – Pointer to spifi_dma_handle_t structure
- callback – SPIFI callback, NULL means no callback.
- userData – User callback function data.
- dmaHandle – User requested DMA handle for DMA transfer

```
void SPIFI_TransferRxCreateHandleDMA(SPIFI_Type *base, spifi_dma_handle_t *handle,  
                                     spifi_dma_callback_t callback, void *userData,  
                                     dma_handle_t *dmaHandle)
```

Initializes the SPIFI handle for receive which is used in transactional functions and set the callback.

Parameters

- base – SPIFI peripheral base address
- handle – Pointer to spifi_dma_handle_t structure
- callback – SPIFI callback, NULL means no callback.
- userData – User callback function data.
- dmaHandle – User requested DMA handle for DMA transfer

```
status_t SPIFI_TransferSendDMA(SPIFI_Type *base, spifi_dma_handle_t *handle, spifi_transfer_t  
                               *xfer)
```

Transfers SPIFI data using an DMA non-blocking method.

This function writes data to the SPIFI transmit FIFO. This function is non-blocking.

Parameters

- base – Pointer to QuadSPI Type.
- handle – Pointer to spifi_dma_handle_t structure

- xfer – SPIFI transfer structure.

status_t SPIFI_TransferReceiveDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *spifi_transfer_t* *xfer)

Receives data using an DMA non-blocking method.

This function receive data from the SPIFI receive buffer/FIFO. This function is non-blocking.

Parameters

- base – Pointer to QuadSPI Type.
- handle – Pointer to *spifi_dma_handle_t* structure
- xfer – SPIFI transfer structure.

void SPIFI_TransferAbortSendDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle)

Aborts the sent data using DMA.

This function aborts the sent data using DMA.

Parameters

- base – SPIFI peripheral base address.
- handle – Pointer to *spifi_dma_handle_t* structure

void SPIFI_TransferAbortReceiveDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle)

Aborts the receive data using DMA.

This function abort receive data which using DMA.

Parameters

- base – SPIFI peripheral base address.
- handle – Pointer to *spifi_dma_handle_t* structure

status_t SPIFI_TransferGetSendCountDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *size_t* *count)

Gets the transferred counts of send.

Parameters

- base – Pointer to QuadSPI Type.
- handle – Pointer to *spifi_dma_handle_t* structure.
- 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 SPIFI_TransferGetReceiveCountDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *size_t* *count)

Gets the status of the receive transfer.

Parameters

- base – Pointer to QuadSPI Type.
- handle – Pointer to *spifi_dma_handle_t* structure
- count – Bytes received.

Return values

- kStatus_Success – Succeed get the transfer count.

- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`uint32_t SPIFI_GetInstance(SPIFI_Type *base)`

Get the SPIFI instance from peripheral base address.

Parameters

- `base` – SPIFI peripheral base address.

Returns

SPIFI instance.

`void SPIFI_Init(SPIFI_Type *base, const spifi_config_t *config)`

Initializes the SPIFI with the user configuration structure.

This function configures the SPIFI module with the user-defined configuration.

Parameters

- `base` – SPIFI peripheral base address.
- `config` – The pointer to the configuration structure.

`void SPIFI_GetDefaultConfig(spifi_config_t *config)`

Get SPIFI default configure settings.

Parameters

- `config` – SPIFI config structure pointer.

`void SPIFI_Deinit(SPIFI_Type *base)`

Deinitializes the SPIFI regions.

Parameters

- `base` – SPIFI peripheral base address.

`void SPIFI_SetCommand(SPIFI_Type *base, spifi_command_t *cmd)`

Set SPIFI flash command.

Parameters

- `base` – SPIFI peripheral base address.
- `cmd` – SPIFI command structure pointer.

`static inline void SPIFI_SetCommandAddress(SPIFI_Type *base, uint32_t addr)`

Set SPIFI command address.

Parameters

- `base` – SPIFI peripheral base address.
- `addr` – Address value for the command.

`static inline void SPIFI_SetIntermediateData(SPIFI_Type *base, uint32_t val)`

Set SPIFI intermediate data.

Before writing a command which needs specific intermediate value, users shall call this function to write it. The main use of this function for current serial flash is to select no-opcode mode and cancelling this mode. As dummy cycle do not care about the value, no need to call this function.

Parameters

- `base` – SPIFI peripheral base address.
- `val` – Intermediate data.

static inline void SPIFI_SetCacheLimit(SPIFI_Type *base, uint32_t val)

Set SPIFI Cache limit value.

SPIFI includes caching of previously-accessed data to improve performance. Software can write an address to this function, to prevent such caching at and above the address.

Parameters

- base – SPIFI peripheral base address.
- val – Zero-based upper limit of cacheable memory.

static inline void SPIFI_ResetCommand(SPIFI_Type *base)

Reset the command field of SPIFI.

This function is used to abort the current command or memory mode.

Parameters

- base – SPIFI peripheral base address.

void SPIFI_SetMemoryCommand(SPIFI_Type *base, *spifi_command_t* *cmd)

Set SPIFI flash AHB read command.

Call this function means SPIFI enters to memory mode, while users need to use command, a SPIFI_ResetCommand shall be called.

Parameters

- base – SPIFI peripheral base address.
- cmd – SPIFI command structure pointer.

static inline void SPIFI_EnableInterrupt(SPIFI_Type *base, uint32_t mask)

Enable SPIFI interrupt.

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

- base – SPIFI peripheral base address.
- mask – SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: *spifi_interrupt_enable_t*

static inline void SPIFI_DisableInterrupt(SPIFI_Type *base, uint32_t mask)

Disable SPIFI interrupt.

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

- base – SPIFI peripheral base address.
- mask – SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: *spifi_interrupt_enable_t*

static inline uint32_t SPIFI_GetStatusFlag(SPIFI_Type *base)

Get the status of all interrupt flags for SPIFI.

Parameters

- base – SPIFI peripheral base address.

Returns

SPIFI flag status

FSL_SPIFI_DMA_DRIVER_VERSION

SPIFI DMA driver version 2.0.3.

FSL_SPIFI_DRIVER_VERSION

SPIFI driver version 2.0.3.

Status structure of SPIFI.

Values:

enumerator kStatus_SPIFI_Idle

SPIFI is in idle state

enumerator kStatus_SPIFI_Busy

SPIFI is busy

enumerator kStatus_SPIFI_Error

Error occurred during SPIFI transfer

enum __spifi_interrupt_enable

SPIFI interrupt source.

Values:

enumerator kSPIFI_CommandFinishInterruptEnable

Interrupt while command finished

enum __spifi_spi_mode

SPIFI SPI mode select.

Values:

enumerator kSPIFI_SPISckLow

SCK low after last bit of command, keeps low while CS high

enumerator kSPIFI_SPISckHigh

SCK high after last bit of command and while CS high

enum __spifi_dual_mode

SPIFI dual mode select.

Values:

enumerator kSPIFI_QuadMode

SPIFI uses IO3:0

enumerator kSPIFI_DualMode

SPIFI uses IO1:0

enum __spifi_data_direction

SPIFI data direction.

Values:

enumerator kSPIFI_DataInput

Data input from serial flash.

enumerator kSPIFI_DataOutput

Data output to serial flash.

enum __spifi_command_format

SPIFI command opcode format.

Values:

enumerator kSPIFI_CommandAllSerial

All fields of command are serial.

enumerator kSPIFI_CommandDataQuad

Only data field is dual/quad, others are serial.

enumerator kSPIFI_CommandOpcodeSerial

Only opcode field is serial, others are quad/dual.

enumerator kSPIFI_CommandAllQuad

All fields of command are dual/quad mode.

enum _spifi_command_type

SPIFI command type.

Values:

enumerator kSPIFI_CommandOpcodeOnly

Command only have opcode, no address field

enumerator kSPIFI_CommandOpcodeAddrOneByte

Command have opcode and also one byte address field

enumerator kSPIFI_CommandOpcodeAddrTwoBytes

Command have opcode and also two bytes address field

enumerator kSPIFI_CommandOpcodeAddrThreeBytes

Command have opcode and also three bytes address field.

enumerator kSPIFI_CommandOpcodeAddrFourBytes

Command have opcode and also four bytes address field

enumerator kSPIFI_CommandNoOpcodeAddrThreeBytes

Command have no opcode and three bytes address field

enumerator kSPIFI_CommandNoOpcodeAddrFourBytes

Command have no opcode and four bytes address field

SPIFI status flags.

Values:

enumerator kSPIFI_MemoryCommandWriteFinished

Memory command write finished

enumerator kSPIFI_CommandWriteFinished

Command write finished

enumerator kSPIFI_InterruptRequest

CMD flag from 1 to 0, means command execute finished

typedef struct *_spifi_dma_handle* spifi_dma_handle_t

typedef void (*spifi_dma_callback_t)(SPIFI_Type *base, *spifi_dma_handle_t* *handle, status_t status, void *userData)

SPIFI DMA transfer callback function for finish and error.

typedef enum *_spifi_interrupt_enable* spifi_interrupt_enable_t

SPIFI interrupt source.

typedef enum *_spifi_spi_mode* spifi_spi_mode_t

SPIFI SPI mode select.

```
typedef enum _spifi_dual_mode spifi_dual_mode_t
```

SPIFI dual mode select.

```
typedef enum _spifi_data_direction spifi_data_direction_t
```

SPIFI data direction.

```
typedef enum _spifi_command_format spifi_command_format_t
```

SPIFI command opcode format.

```
typedef enum _spifi_command_type spifi_command_type_t
```

SPIFI command type.

```
typedef struct _spifi_command spifi_command_t
```

SPIFI command structure.

```
typedef struct _spifi_config spifi_config_t
```

SPIFI region configuration structure.

```
typedef struct _spifi_transfer spifi_transfer_t
```

Transfer structure for SPIFI.

```
static inline void SPIFI_EnableDMA(SPIFI_Type *base, bool enable)
```

Enable or disable DMA request for SPIFI.

Parameters

- base – SPIFI peripheral base address.
- enable – True means enable DMA and false means disable DMA.

```
static inline uint32_t SPIFI_GetDataRegisterAddress(SPIFI_Type *base)
```

Gets the SPIFI data register address.

This API is used to provide a transfer address for the SPIFI DMA transfer configuration.

Parameters

- base – SPIFI base pointer

Returns

data register address

```
static inline void SPIFI_WriteData(SPIFI_Type *base, uint32_t data)
```

Write a word data in address of SPIFI.

Users can write a page or at least a word data into SPIFI address.

Parameters

- base – SPIFI peripheral base address.
- data – Data need be write.

```
static inline void SPIFI_WriteDataByte(SPIFI_Type *base, uint8_t data)
```

Write a byte data in address of SPIFI.

Users can write a byte data into SPIFI address.

Parameters

- base – SPIFI peripheral base address.
- data – Data need be write.

```
void SPIFI_WriteDataHalfword(SPIFI_Type *base, uint16_t data)
```

Write a halfword data in address of SPIFI.

Users can write a halfword data into SPIFI address.

Parameters

- base – SPIFI peripheral base address.
- data – Data need be write.

static inline uint32_t SPIFI_ReadData(SPIFI_Type *base)

Read data from serial flash.

Users should notice before call this function, the data length field in command register shall larger than 4, otherwise a hardfault will happen.

Parameters

- base – SPIFI peripheral base address.

Returns

Data input from flash.

static inline uint8_t SPIFI_ReadDataByte(SPIFI_Type *base)

Read a byte data from serial flash.

Parameters

- base – SPIFI peripheral base address.

Returns

Data input from flash.

uint16_t SPIFI_ReadDataHalfword(SPIFI_Type *base)

Read a halfword data from serial flash.

Parameters

- base – SPIFI peripheral base address.

Returns

Data input from flash.

struct _spifi_dma_handle

#include <fsl_spifi_dma.h> SPIFI DMA transfer handle, users should not touch the content of the handle.

Public Members

dma_handle_t *dmaHandle

DMA handler for SPIFI send

size_t transferSize

Bytes need to transfer.

uint32_t state

Internal state for SPIFI DMA transfer

spifi_dma_callback_t callback

Callback for users while transfer finish or error occurred

void *userData

User callback parameter

struct _spifi_command

#include <fsl_spifi.h> SPIFI command structure.

Public Members

`uint16_t dataLen`
How many data bytes are needed in this command.

`bool isPollMode`
For command need to read data from serial flash

`spifi_data_direction_t direction`
Data direction of this command.

`uint8_t intermediateBytes`
How many intermediate bytes needed

`spifi_command_format_t format`
Command format

`spifi_command_type_t type`
Command type

`uint8_t opcode`
Command opcode value

`struct __spifi_config`
`#include <fsl_spifi.h>` SPIFI region configuration structure.

Public Members

`uint16_t timeout`
SPI transfer timeout, the unit is SCK cycles

`uint8_t csHighTime`
CS high time cycles

`bool disablePrefetch`
True means SPIFI will not attempt a speculative prefetch.

`bool disableCachePrefech`
Disable prefetch of cache line

`bool isFeedbackClock`
Is data sample uses feedback clock.

`spifi_spi_mode_t spiMode`
SPIFI spi mode select

`bool isReadFullClockCycle`
If enable read full clock cycle.

`spifi_dual_mode_t dualMode`
SPIFI dual mode, dual or quad.

`struct __spifi_transfer`
`#include <fsl_spifi.h>` Transfer structure for SPIFI.

Public Members

`uint8_t *data`
Pointer to data to transmit

`size_t dataSize`
Bytes to be transmit

2.38 SPIFI DMA Driver

2.39 SPIFI Driver

2.40 TRNG: True Random Number Generator

FSL_TRNG_DRIVER_VERSION

TRNG driver version 2.0.18.

Current version: 2.0.18

Change log:

- version 2.0.18
 - TRNG health checks now done in software on RT5xx and RT6xx.
- version 2.0.17
 - Added support for RT700.
- version 2.0.16
 - Added support for Dual oscillator mode.
- version 2.0.15
 - Changed TRNG_USER_CONFIG_DEFAULT_XXX values according to latest recommended by design team.
- version 2.0.14
 - add support for RW610 and RW612
- version 2.0.13
 - After deepsleep it might return error, added clearing bits in TRNG_GetRandomData() and generating new entropy.
 - Modified reloading entropy in TRNG_GetRandomData(), for some data length it doesn't reloading entropy correctly.
- version 2.0.12
 - For KW34A4_SERIES, KW35A4_SERIES, KW36A4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.
- version 2.0.11
 - Add clearing pending errors in TRNG_Init().
- version 2.0.10
 - Fixed doxygen issues.
- version 2.0.9
 - Fix HIS_CCM metrics issues.
- version 2.0.8
 - For K32L2A41A_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv4.
- version 2.0.7
 - Fix MISRA 2004 issue rule 12.5.
- version 2.0.6

- For KW35Z4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.
- version 2.0.5
 - Add possibility to define default TRNG configuration by device specific preprocessor macros for FRQMIN, FRQMAX and OSCDIV.
- version 2.0.4
 - Fix MISRA-2012 issues.
- Version 2.0.3
 - update TRNG_Init to restart entropy generation
- Version 2.0.2
 - fix MISRA issues
- Version 2.0.1
 - add support for KL8x and KL28Z
 - update default OSCDIV for K81 to divide by 2

enum __trng_sample_mode

TRNG sample mode. Used by trng_config_t.

Values:

enumerator kTRNG_SampleModeVonNeumann

Use von Neumann data in both Entropy shifter and Statistical Checker.

enumerator kTRNG_SampleModeRaw

Use raw data into both Entropy shifter and Statistical Checker.

enumerator kTRNG_SampleModeVonNeumannRaw

Use von Neumann data in Entropy shifter. Use raw data into Statistical Checker.

enum __trng_clock_mode

TRNG clock mode. Used by trng_config_t.

Values:

enumerator kTRNG_ClockModeRingOscillator

Ring oscillator is used to operate the TRNG (default).

enumerator kTRNG_ClockModeSystem

System clock is used to operate the TRNG. This is for test use only, and indeterminate results may occur.

enum __trng_ring_osc_div

TRNG ring oscillator divide. Used by trng_config_t.

Values:

enumerator kTRNG_RingOscDiv0

Ring oscillator with no divide

enumerator kTRNG_RingOscDiv2

Ring oscillator divided-by-2.

enumerator kTRNG_RingOscDiv4

Ring oscillator divided-by-4.

enumerator kTRNG_RingOscDiv8

Ring oscillator divided-by-8.

typedef enum *_trng_sample_mode* trng_sample_mode_t

TRNG sample mode. Used by trng_config_t.

typedef enum *_trng_clock_mode* trng_clock_mode_t

TRNG clock mode. Used by trng_config_t.

typedef enum *_trng_ring_osc_div* trng_ring_osc_div_t

TRNG ring oscillator divide. Used by trng_config_t.

typedef struct *_trng_statistical_check_limit* trng_statistical_check_limit_t

Data structure for definition of statistical check limits. Used by trng_config_t.

typedef struct *_trng_user_config* trng_config_t

Data structure for the TRNG initialization.

This structure initializes the TRNG by calling the TRNG_Init() function. It contains all TRNG configurations.

status_t TRNG_GetDefaultConfig(*trng_config_t* *userConfig)

Initializes the user configuration structure to default values.

This function initializes the configuration structure to default values. The default values are platform dependent.

Parameters

- userConfig – User configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

status_t TRNG_Init(TRNG_Type *base, const *trng_config_t* *userConfig)

Initializes the TRNG.

This function initializes the TRNG. When called, the TRNG entropy generation starts immediately.

Parameters

- base – TRNG base address
- userConfig – Pointer to the initialization configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

void TRNG_Deinit(TRNG_Type *base)

Shuts down the TRNG.

This function shuts down the TRNG.

Parameters

- base – TRNG base address.

status_t TRNG_GetRandomData(TRNG_Type *base, void *data, size_t dataSize)

Gets random data.

This function gets random data from the TRNG.

Parameters

- base – TRNG base address.
- data – Pointer address used to store random data.
- dataSize – Size of the buffer pointed by the data parameter.

Returns

random data

struct `_trng_statistical_check_limit`

#include <fsl_trng.h> Data structure for definition of statistical check limits. Used by `trng_config_t`.

Public Members

uint32_t maximum

Maximum limit.

int32_t minimum

Minimum limit.

struct `_trng_user_config`

#include <fsl_trng.h> Data structure for the TRNG initialization.

This structure initializes the TRNG by calling the `TRNG_Init()` function. It contains all TRNG configurations.

Public Members

bool lock

Disable programmability of TRNG registers.

trng_clock_mode_t clockMode

Clock mode used to operate TRNG.

trng_ring_osc_div_t ringOscDiv

Ring oscillator divide used by TRNG.

trng_sample_mode_t sampleMode

Sample mode of the TRNG ring oscillator.

uint16_t entropyDelay

Entropy Delay. Defines the length (in system clocks) of each Entropy sample taken.

uint16_t sampleSize

Sample Size. Defines the total number of Entropy samples that will be taken during Entropy generation.

uint16_t sparseBitLimit

Sparse Bit Limit which defines the maximum number of consecutive samples that may be discarded before an error is generated. This limit is used only for during von Neumann sampling (enabled by `TRNG_HAL_SetSampleMode()`). Samples are discarded if two consecutive raw samples are both 0 or both 1. If this discarding occurs for a long period of time, it indicates that there is insufficient Entropy.

uint8_t retryCount

Retry count. It defines the number of times a statistical check may fails during the TRNG Entropy Generation before generating an error.

uint8_t longRunMaxLimit

Largest allowable number of consecutive samples of all 1, or all 0, that is allowed during the Entropy generation.

trng_statistical_check_limit_t monobitLimit

Maximum and minimum limits for statistical check of number of ones/zero detected during entropy generation.

trng_statistical_check_limit_t runBit1Limit

Maximum and minimum limits for statistical check of number of runs of length 1 detected during entropy generation.

trng_statistical_check_limit_t runBit2Limit

Maximum and minimum limits for statistical check of number of runs of length 2 detected during entropy generation.

trng_statistical_check_limit_t runBit3Limit

Maximum and minimum limits for statistical check of number of runs of length 3 detected during entropy generation.

trng_statistical_check_limit_t runBit4Limit

Maximum and minimum limits for statistical check of number of runs of length 4 detected during entropy generation.

trng_statistical_check_limit_t runBit5Limit

Maximum and minimum limits for statistical check of number of runs of length 5 detected during entropy generation.

trng_statistical_check_limit_t runBit6PlusLimit

Maximum and minimum limits for statistical check of number of runs of length 6 or more detected during entropy generation.

trng_statistical_check_limit_t pokerLimit

Maximum and minimum limits for statistical check of “Poker Test”.

trng_statistical_check_limit_t frequencyCountLimit

Maximum and minimum limits for statistical check of entropy sample frequency count.

2.41 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.42 USART DMA Driver

status_t USART_TransferCreateHandleDMA(USART_Type *base, *usart_dma_handle_t* *handle, *usart_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *txDmaHandle, *dma_handle_t* *rxDmaHandle)

Initializes the USART handle which is used in transactional functions.

Parameters

- base – USART peripheral base address.
- handle – Pointer to *usart_dma_handle_t* structure.
- callback – Callback function.
- userData – User data.
- txDmaHandle – User-requested DMA handle for TX DMA transfer.
- rxDmaHandle – User-requested DMA handle for RX DMA transfer.

```
status_t USART_TransferSendDMA(USART_Type *base, usart_dma_handle_t *handle,  
                               usart_transfer_t *xfer)
```

Sends data using DMA.

This function sends data using DMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_TxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
status_t USART_TransferReceiveDMA(USART_Type *base, usart_dma_handle_t *handle,  
                                  usart_transfer_t *xfer)
```

Receives data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – USART peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_RxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
void USART_TransferAbortSendDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the sent data using DMA.
```

This function aborts send data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure

```
void USART_TransferAbortReceiveDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the received data using DMA.
```

This function aborts the received data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure


```
status_t USART_TransferGetReceiveCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                           uint32_t *count)
```

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
status_t USART_TransferGetSendCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                        uint32_t *count)
```

Get the number of bytes that have been sent.

This function gets the number of bytes that have been sent.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Sent bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
FSL_USART_DMA_DRIVER_VERSION
```

USART dma driver version.

```
typedef struct _usart_dma_handle usart_dma_handle_t
```

```
typedef void (*usart_dma_transfer_callback_t)(USART_Type *base, usart_dma_handle_t *handle,
status_t status, void *userData)
```

UART transfer callback function.

```
struct _usart_dma_handle
```

```
#include <fsl_usart_dma.h> UART DMA handle.
```

Public Members

```
USART_Type *base
```

UART peripheral base address.

```
usart_dma_transfer_callback_t callback
```

Callback function.

```
void *userData
```

UART callback function parameter.

`size_t rxDataSizeAll`
Size of the data to receive.

`size_t txDataSizeAll`
Size of the data to send out.

`dma_handle_t *txDmaHandle`
The DMA TX channel used.

`dma_handle_t *rxDmaHandle`
The DMA RX channel used.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state

2.43 USART Driver

`status_t USART__Init(USART_Type *base, const usart_config_t *config, uint32_t srcClock_Hz)`
Initializes a USART instance with user configuration structure and peripheral clock.

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the `USART_GetDefaultConfig()` function. Example below shows how to use this API to configure USART.

```
usart_config_t usartConfig;  
usartConfig.baudRate_Bps = 115200U;  
usartConfig.parityMode = kUSART_ParityDisabled;  
usartConfig.stopBitCount = kUSART_OneStopBit;  
USART__Init(USART1, &usartConfig, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `config` – Pointer to user-defined configuration structure.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_InvalidArgument` – USART base address is not valid
- `kStatus_Success` – Status USART initialize succeed

`void USART__Deinit(USART_Type *base)`

Deinitializes a USART instance.

This function waits for TX complete, disables TX and RX, and disables the USART clock.

Parameters

- `base` – USART peripheral base address.

```
void USART_GetDefaultConfig(usart_config_t *config)
```

Gets the default configuration structure.

This function initializes the USART configuration structure to a default value. The default values are: `usartConfig->baudRate_Bps = 115200U`; `usartConfig->parityMode = kUSART_ParityDisabled`; `usartConfig->stopBitCount = kUSART_OneStopBit`; `usartConfig->bitCountPerChar = kUSART_8BitsPerChar`; `usartConfig->loopback = false`; `usartConfig->enableTx = false`; `usartConfig->enableRx = false`;

Parameters

- `config` – Pointer to configuration structure.

```
status_t USART_SetBaudRate(USART_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
```

Sets the USART instance baud rate.

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the `USART_Init`.

```
USART_SetBaudRate(USART1, 115200U, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `baudrate_Bps` – USART baudrate to be set.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

```
status_t USART_Enable32kMode(USART_Type *base, uint32_t baudRate_Bps, bool  
enableMode32k, uint32_t srcClock_Hz)
```

Enable 32 kHz mode which USART uses clock from the RTC oscillator as the clock source.

Please note that in order to use a 32 kHz clock to operate USART properly, the RTC oscillator and its 32 kHz output must be manually enabled by user, by calling `RTC_Init` and setting `SYSCON_RTCOSCTRL_EN` bit to 1. And in 32kHz clocking mode the USART can only work at 9600 baudrate or at the baudrate that 9600 can evenly divide, eg: 4800, 3200.

Parameters

- `base` – USART peripheral base address.
- `baudRate_Bps` – USART baudrate to be set..
- `enableMode32k` – true is 32k mode, false is normal mode.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

void USART_Enable9bitMode(USART_Type *base, bool enable)

Enable 9-bit data mode for USART.

This function set the 9-bit mode for USART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – USART peripheral base address.
- enable – true to enable, false to disable.

static inline void USART_SetMatchAddress(USART_Type *base, uint8_t address)

Set the USART slave address.

This function configures the address for USART module that works as slave in 9-bit data mode. When the address detection is enabled, 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 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 USART 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 – USART peripheral base address.
- address – USART slave address.

static inline void USART_EnableMatchAddress(USART_Type *base, bool match)

Enable the USART match address feature.

Parameters

- base – USART peripheral base address.
- match – true to enable match address, false to disable.

static inline uint32_t USART_GetStatusFlags(USART_Type *base)

Get USART status flags.

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators `_usart_flags`. To check a specific status, compare the return value with enumerators in `_usart_flags`. For example, to check whether the TX is empty:

```
if (kUSART_TxFifoNotFullFlag & USART_GetStatusFlags(USART1))
{
    ...
}
```

Parameters

- base – USART peripheral base address.

Returns

USART status flags which are ORed by the enumerators in the `_usart_flags`.

static inline void USART_ClearStatusFlags(USART_Type *base, uint32_t mask)

Clear USART status flags.

This function clear supported USART status flags. The mask is a logical OR of enumeration members. See `kUSART_AllClearFlags`. For example:

```
USART_ClearStatusFlags(USART1, kUSART_TxError | kUSART_RxError)
```

Parameters

- base – USART peripheral base address.
- mask – status flags to be cleared.

```
static inline void USART_EnableInterrupts(USART_Type *base, uint32_t mask)
```

Enables USART interrupts according to the provided mask.

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. For example, to enable TX empty interrupt and RX full interrupt:

```
USART_EnableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART_
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_usart_interrupt_enable`.

```
static inline void USART_DisableInterrupts(USART_Type *base, uint32_t mask)
```

Disables USART interrupts according to a provided mask.

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
USART_DisableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART_
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to disable. Logical OR of `_usart_interrupt_enable`.

```
static inline uint32_t USART_GetEnabledInterrupts(USART_Type *base)
```

Returns enabled USART interrupts.

This function returns the enabled USART interrupts.

Parameters

- base – USART peripheral base address.

```
static inline void USART_EnableTxDMA(USART_Type *base, bool enable)
```

Enable DMA for Tx.

```
static inline void USART_EnableRxDMA(USART_Type *base, bool enable)
```

Enable DMA for Rx.

```
static inline void USART_EnableCTS(USART_Type *base, bool enable)
```

Enable CTS. This function will determine whether CTS is used for flow control.

Parameters

- base – USART peripheral base address.
- enable – Enable CTS or not, true for enable and false for disable.

static inline void USART__EnableContinuousSCLK(USART_Type *base, bool enable)

Continuous Clock generation. By default, SCLK is only output while data is being transmitted in synchronous mode. Enable this function, SCLK will run continuously in synchronous mode, allowing characters to be received on Un_RxD independently from transmission on Un_TXD).

Parameters

- base – USART peripheral base address.
- enable – Enable Continuous Clock generation mode or not, true for enable and false for disable.

static inline void USART__EnableAutoClearSCLK(USART_Type *base, bool enable)

Enable Continuous Clock generation bit auto clear. While enable this function, the Continuous Clock bit is automatically cleared when a complete character has been received. This bit is cleared at the same time.

Parameters

- base – USART peripheral base address.
- enable – Enable auto clear or not, true for enable and false for disable.

static inline void USART__SetRxFifoWatermark(USART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Rx FIFO watermark.

static inline void USART__SetTxFifoWatermark(USART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Tx FIFO watermark.

static inline void USART__WriteByte(USART_Type *base, uint8_t data)

Writes to the FIFOWR register.

This function writes data to the txFIFO directly. The upper layer must ensure that txFIFO has space for data to write before calling this function.

Parameters

- base – USART peripheral base address.
- data – The byte to write.

static inline uint8_t USART__ReadByte(USART_Type *base)

Reads the FIFORD register directly.

This function reads data from the rxFIFO directly. The upper layer must ensure that the rxFIFO is not empty before calling this function.

Parameters

- base – USART peripheral base address.

Returns

The byte read from USART data register.

```
static inline uint8_t USART_GetRxFifoCount(USART_Type *base)
```

Gets the rx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

rx FIFO data count.

```
static inline uint8_t USART_GetTxFifoCount(USART_Type *base)
```

Gets the tx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

tx FIFO data count.

```
void USART_SendAddress(USART_Type *base, uint8_t address)
```

Transmit an address frame in 9-bit data mode.

Parameters

- base – USART peripheral base address.
- address – USART slave address.

```
status_t USART_WriteBlocking(USART_Type *base, const uint8_t *data, size_t length)
```

Writes to the TX register using a blocking method.

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Parameters

- base – USART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_USART_Timeout – Transmission timed out and was aborted.
- kStatus_InvalidArgument – Invalid argument.
- kStatus_Success – Successfully wrote all data.

```
status_t USART_ReadBlocking(USART_Type *base, uint8_t *data, size_t length)
```

Read RX data register using a blocking method.

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data and read data from the TX register.

Parameters

- base – USART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_USART_FramingError – Receiver overrun happened while receiving data.
- kStatus_USART_ParityError – Noise error happened while receiving data.

- `kStatus_USART_NoiseError` – Framing error happened while receiving data.
- `kStatus_USART_RxError` – Overflow or underflow rxFIFO happened.
- `kStatus_USART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

`status_t` `USART_TransferCreateHandle`(`USART_Type` *base, `usart_handle_t` *handle, `usart_transfer_callback_t` callback, void *userData)

Initializes the USART handle.

This function initializes the USART handle which can be used for other USART transactional APIs. Usually, for a specified USART instance, call this API once to get the initialized handle.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

`status_t` `USART_TransferSendNonBlocking`(`USART_Type` *base, `usart_handle_t` *handle, `usart_transfer_t` *xfer)

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the IRQ handler, the USART driver calls the callback function and passes the `kStatus_USART_TxIdle` as status parameter.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_USART_TxBusy` – Previous transmission still not finished, data not all written to TX register yet.
- `kStatus_InvalidArgument` – Invalid argument.

`void` `USART_TransferStartRingBuffer`(`USART_Type` *base, `usart_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 USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the `USART_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 the 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 – USART peripheral base address.

- `handle` – USART handle pointer.
- `ringBuffer` – Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
- `ringBufferSize` – size of the ring buffer.

`void USART_TransferStopRingBuffer(USART_Type *base, usart_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` – USART peripheral base address.
- `handle` – USART handle pointer.

`size_t USART_TransferGetRxRingBufferLength(usart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `handle` – USART handle pointer.

Returns

Length of received data in RX ring buffer.

`void USART_TransferAbortSend(USART_Type *base, usart_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 still not sent out.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

`status_t USART_TransferGetSendCount(USART_Type *base, usart_handle_t *handle, uint32_t *count)`

Get 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 interrupt method.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART 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 USART_TransferReceiveNonBlocking(USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer, size_t *receivedBytes)`

Receives a buffer of data using an interrupt method.

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the USART driver. When the new

data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_USART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the USART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.
- `xfer` – USART transfer structure, see `usart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into transmit queue.
- `kStatus_USART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void USART_TransferAbortReceive(USART_Type *base, usart_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` – USART peripheral base address.
- `handle` – USART handle pointer.

`status_t USART_TransferGetReceiveCount(USART_Type *base, usart_handle_t *handle, uint32_t *count)`

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART 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 USART_TransferHandleIRQ(USART_Type *base, usart_handle_t *handle)`

USART IRQ handle function.

This function handles the USART transmit and receive IRQ request.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

FSL_USART_DRIVER_VERSION

USART driver version.

Error codes for the USART driver.

Values:

enumerator kStatus_USART_TxBusy

Transmitter is busy.

enumerator kStatus_USART_RxBusy

Receiver is busy.

enumerator kStatus_USART_TxIdle

USART transmitter is idle.

enumerator kStatus_USART_RxIdle

USART receiver is idle.

enumerator kStatus_USART_TxError

Error happens on txFIFO.

enumerator kStatus_USART_RxError

Error happens on rxFIFO.

enumerator kStatus_USART_RxRingBufferOverrun

Error happens on rx ring buffer

enumerator kStatus_USART_NoiseError

USART noise error.

enumerator kStatus_USART_FramingError

USART framing error.

enumerator kStatus_USART_ParityError

USART parity error.

enumerator kStatus_USART_BaudrateNotSupport

Baudrate is not support in current clock source

enum _usart_sync_mode

USART synchronous mode.

Values:

enumerator kUSART_SyncModeDisabled

Asynchronous mode.

enumerator kUSART_SyncModeSlave

Synchronous slave mode.

enumerator kUSART_SyncModeMaster

Synchronous master mode.

enum _usart_parity_mode

USART parity mode.

Values:

enumerator kUSART_ParityDisabled

Parity disabled

enumerator kUSART_ParityEven

Parity enabled, type even, bit setting: PE | PT = 10

enumerator kUSART_ParityOdd

Parity enabled, type odd, bit setting: PE | PT = 11

enum _usart_stop_bit_count

USART stop bit count.

Values:

enumerator kUSART_OneStopBit

One stop bit

enumerator kUSART_TwoStopBit

Two stop bits

enum _usart_data_len

USART data size.

Values:

enumerator kUSART_7BitsPerChar

Seven bit mode

enumerator kUSART_8BitsPerChar

Eight bit mode

enum _usart_clock_polarity

USART clock polarity configuration, used in sync mode.

Values:

enumerator kUSART_RxSampleOnFallingEdge

Un_RXD is sampled on the falling edge of SCLK.

enumerator kUSART_RxSampleOnRisingEdge

Un_RXD is sampled on the rising edge of SCLK.

enum _usart_txfifo_watermark

txFIFO watermark values

Values:

enumerator kUSART_TxFifo0

USART tx watermark is empty

enumerator kUSART_TxFifo1

USART tx watermark at 1 item

enumerator kUSART_TxFifo2

USART tx watermark at 2 items

enumerator kUSART_TxFifo3

USART tx watermark at 3 items

enumerator kUSART_TxFifo4

USART tx watermark at 4 items

enumerator kUSART_TxFifo5

USART tx watermark at 5 items

enumerator kUSART_TxFifo6

USART tx watermark at 6 items

enumerator kUSART_TxFifo7
USART tx watermark at 7 items

enum __usart_rxfifo_watermark
rxFIFO watermark values

Values:

enumerator kUSART_RxFifo1
USART rx watermark at 1 item

enumerator kUSART_RxFifo2
USART rx watermark at 2 items

enumerator kUSART_RxFifo3
USART rx watermark at 3 items

enumerator kUSART_RxFifo4
USART rx watermark at 4 items

enumerator kUSART_RxFifo5
USART rx watermark at 5 items

enumerator kUSART_RxFifo6
USART rx watermark at 6 items

enumerator kUSART_RxFifo7
USART rx watermark at 7 items

enumerator kUSART_RxFifo8
USART rx watermark at 8 items

enum __usart_interrupt_enable
USART interrupt configuration structure, default settings all disabled.

Values:

enumerator kUSART_TxErrorInterruptEnable

enumerator kUSART_RxErrorInterruptEnable

enumerator kUSART_TxLevelInterruptEnable

enumerator kUSART_RxLevelInterruptEnable

enumerator kUSART_TxIdleInterruptEnable
Transmitter idle.

enumerator kUSART_CtsChangeInterruptEnable
Change in the state of the CTS input.

enumerator kUSART_RxBreakChangeInterruptEnable
Break condition asserted or deasserted.

enumerator kUSART_RxStartInterruptEnable
Rx start bit detected.

enumerator kUSART_FramingErrorInterruptEnable
Framing error detected.

enumerator kUSART_ParityErrorInterruptEnable
Parity error detected.

enumerator kUSART_NoiseErrorInterruptEnable

Noise error detected.

enumerator kUSART_AutoBaudErrorInterruptEnable

Auto baudrate error detected.

enumerator kUSART_AllInterruptEnables

enum __usart_flags

USART status flags.

This provides constants for the USART status flags for use in the USART functions.

Values:

enumerator kUSART_TxError

TXERR bit, sets if TX buffer is error

enumerator kUSART_RxError

RXERR bit, sets if RX buffer is error

enumerator kUSART_TxFifoEmptyFlag

TXEMPTY bit, sets if TX buffer is empty

enumerator kUSART_TxFifoNotFullFlag

TXNOTFULL bit, sets if TX buffer is not full

enumerator kUSART_RxFifoNotEmptyFlag

RXNOEMPTY bit, sets if RX buffer is not empty

enumerator kUSART_RxFifoFullFlag

RXFULL bit, sets if RX buffer is full

enumerator kUSART_RxIdleFlag

Receiver idle.

enumerator kUSART_TxIdleFlag

Transmitter idle.

enumerator kUSART_CtsAssertFlag

CTS signal high.

enumerator kUSART_CtsChangeFlag

CTS signal changed interrupt status.

enumerator kUSART_BreakDetectFlag

Break detected. Self cleared when rx pin goes high again.

enumerator kUSART_BreakDetectChangeFlag

Break detect change interrupt flag. A change in the state of receiver break detection.

enumerator kUSART_RxStartFlag

Rx start bit detected interrupt flag.

enumerator kUSART_FramingErrorFlag

Framing error interrupt flag.

enumerator kUSART_ParityErrorFlag

parity error interrupt flag.

enumerator kUSART_NoiseErrorFlag

Noise error interrupt flag.

enumerator `kUSART_AutobaudErrorFlag`

Auto baudrate error interrupt flag, caused by the baudrate counter timeout before the end of start bit.

enumerator `kUSART_AllClearFlags`

typedef enum `_usart_sync_mode` `usart_sync_mode_t`

USART synchronous mode.

typedef enum `_usart_parity_mode` `usart_parity_mode_t`

USART parity mode.

typedef enum `_usart_stop_bit_count` `usart_stop_bit_count_t`

USART stop bit count.

typedef enum `_usart_data_len` `usart_data_len_t`

USART data size.

typedef enum `_usart_clock_polarity` `usart_clock_polarity_t`

USART clock polarity configuration, used in sync mode.

typedef enum `_usart_txfifo_watermark` `usart_txfifo_watermark_t`

txFIFO watermark values

typedef enum `_usart_rxfifo_watermark` `usart_rxfifo_watermark_t`

rxFIFO watermark values

typedef struct `_usart_config` `usart_config_t`

USART configuration structure.

typedef struct `_usart_transfer` `usart_transfer_t`

USART transfer structure.

typedef struct `_usart_handle` `usart_handle_t`

typedef void (*`usart_transfer_callback_t`)(`USART_Type` *`base`, `usart_handle_t` *`handle`, `status_t` `status`, void *`userData`)

USART transfer callback function.

typedef void (*`flexcomm_usart_irq_handler_t`)(`USART_Type` *`base`, `usart_handle_t` *`handle`)

Typedef for usart interrupt handler.

uint32_t `USART_GetInstance(USART_Type *base)`

Returns instance number for USART peripheral base address.

`USART_FIFOTRIG_TXLVL_GET(base)`

`USART_FIFOTRIG_RXLVL_GET(base)`

`UART_RETRY_TIMES`

Retry times for waiting flag.

Defining to zero means to keep waiting for the flag until it is assert/deassert in blocking transfer; otherwise the program will wait until the `UART_RETRY_TIMES` counts down to 0, if the flag still remains unchanged then program will return `kStatus_USART_Timeout`. It is not advised to use this macro in formal application to prevent any hardware error because the actual wait period is affected by the compiler and optimization.

struct `_usart_config`

`#include <fsl_usart.h>` USART configuration structure.

Public Members

`uint32_t` baudRate_Bps
USART baud rate

`usart_parity_mode_t` parityMode
Parity mode, disabled (default), even, odd

`usart_stop_bit_count_t` stopBitCount
Number of stop bits, 1 stop bit (default) or 2 stop bits

`usart_data_len_t` bitCountPerChar
Data length - 7 bit, 8 bit

`bool` loopback
Enable peripheral loopback

`bool` enableRx
Enable RX

`bool` enableTx
Enable TX

`bool` enableContinuousSCLK
USART continuous Clock generation enable in synchronous master mode.

`bool` enableMode32k
USART uses 32 kHz clock from the RTC oscillator as the clock source.

`bool` enableHardwareFlowControl
Enable hardware control RTS/CTS

`usart_txfifo_watermark_t` txWatermark
txFIFO watermark

`usart_rxfifo_watermark_t` rxWatermark
rxFIFO watermark

`usart_sync_mode_t` syncMode
Transfer mode select - asynchronous, synchronous master, synchronous slave.

`usart_clock_polarity_t` clockPolarity
Selects the clock polarity and sampling edge in synchronous mode.

`struct __usart__transfer`
#include <fsl_usart.h> USART transfer structure.

Public Members

`size_t` dataSize
The byte count to be transfer.

`struct __usart__handle`
#include <fsl_usart.h> USART handle structure.

Public Members

`const uint8_t *volatile` txData
Address of remaining data to send.

`volatile size_t txDataSize`
Size of the remaining data to send.

`size_t txDataSizeAll`
Size of the data to send out.

`uint8_t *volatile rxData`
Address of remaining data to receive.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t rxDataSizeAll`
Size of the data to receive.

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

`volatile uint16_t rxRingBufferTail`
Index for the user to get data from the ring buffer.

`usart_transfer_callback_t callback`
Callback function.

`void *userData`
USART callback function parameter.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state

`uint8_t txWatermark`
txFIFO watermark

`uint8_t rxWatermark`
rxFIFO watermark

`union __unnamed13__`

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.

2.44 UTICK: Mictotick Timer Driver

`void UTICK_Init(UTICK_Type *base)`

Initializes an UTICK by turning its bus clock on.

`void UTICK_Deinit(UTICK_Type *base)`

Deinitializes a UTICK instance.

This function shuts down Utick bus clock

Parameters

- `base` – UTICK peripheral base address.

`uint32_t UTICK_GetStatusFlags(UTICK_Type *base)`

Get Status Flags.

This returns the status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

status register value

`void UTICK_ClearStatusFlags(UTICK_Type *base)`

Clear Status Interrupt Flags.

This clears intr status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

none

`void UTICK_SetTick(UTICK_Type *base, utick_mode_t mode, uint32_t count, utick_callback_t cb)`

Starts UTICK.

This function starts a repeat/onetime countdown with an optional callback

Parameters

- `base` – UTICK peripheral base address.
- `mode` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `count` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `cb` – UTICK callback (can be left as `NULL` if none, otherwise should be a `void func(void)`)

Returns

none

`void UTICK_HandleIRQ(UTICK_Type *base, utick_callback_t cb)`

UTICK Interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in `UTICK_SetTick()`). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- `base` – UTICK peripheral base address.

- `cb` – callback scheduled for this instance of UTICK

Returns

none

FSL_UTICK_DRIVER_VERSION

UTICK driver version 2.0.5.

enum `_utick_mode`

UTICK timer operational mode.

*Values:*enumerator `kUTICK_Onetime`

Trigger once

enumerator `kUTICK_Repeat`

Trigger repeatedly

typedef enum `_utick_mode` `utick_mode_t`

UTICK timer operational mode.

typedef void (`*utick_callback_t`)(void)

UTICK callback function.

2.45 WWDT: Windowed Watchdog Timer Driver

void WWDT_GetDefaultConfig(`wwdt_config_t` *config)

Initializes WWDT configure structure.

This function initializes the WWDT configure structure to default value. The default value are:

```
config->enableWwdt = true;
config->enableWatchdogReset = false;
config->enableWatchdogProtect = false;
config->enableLockOscillator = false;
config->windowValue = 0xFFFFFfU;
config->timeoutValue = 0xFFFFFfU;
config->warningValue = 0;
```

See also:`wwdt_config_t`**Parameters**

- `config` – Pointer to WWDT config structure.

void WWDT_Init(WWDT_Type *base, const `wwdt_config_t` *config)

Initializes the WWDT.

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
wwdt_config_t config;
WWDT_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
WWDT_Init(wwdt_base,&config);
```

Parameters

- base – WWDT peripheral base address
- config – The configuration of WWDT

void WWDT_Deinit(WWDT_Type *base)

Shuts down the WWDT.

This function shuts down the WWDT.

Parameters

- base – WWDT peripheral base address

static inline void WWDT_Enable(WWDT_Type *base)

Enables the WWDT module.

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

- base – WWDT peripheral base address

static inline void WWDT_Disable(WWDT_Type *base)

Disables the WWDT module.

Deprecated:

Do not use this function. It will be deleted in next release version, for once the bit field of WDEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

- base – WWDT peripheral base address

static inline uint32_t WWDT_GetStatusFlags(WWDT_Type *base)

Gets all WWDT status flags.

This function gets all status flags.

Example for getting Timeout Flag:

```
uint32_t status;  
status = WWDT_GetStatusFlags(wwdt_base) & kWWDT_TimeoutFlag;
```

Parameters

- base – WWDT peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `_wwdt_status_flags_t`

void WWDT_ClearStatusFlags(WWDT_Type *base, uint32_t mask)

Clear WWDT flag.

This function clears WWDT status flag.

Example for clearing warning flag:

```
WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
```

Parameters

- `base` – WWDT peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `_wwdt_status_flags_t`

`static inline void WWDT_SetWarningValue(WWDT_Type *base, uint32_t warningValue)`

Set the WWDT warning value.

The `WDWARNINT` register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by `WARNINT`, an interrupt will be generated after the subsequent `WDCLK`.

Parameters

- `base` – WWDT peripheral base address
- `warningValue` – WWDT warning value.

`static inline void WWDT_SetTimeoutValue(WWDT_Type *base, uint32_t timeoutCount)`

Set the WWDT timeout value.

This function sets the timeout value. Every time a feed sequence occurs the value in the `TC` register is loaded into the Watchdog timer. Writing a value below `0xFF` will cause `0xFF` to be loaded into the `TC` register. Thus the minimum time-out interval is $TWDCLK * 256 * 4$. If `enableWatchdogProtect` flag is true in `wwdt_config_t` config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the `WDTOF` flag.

Parameters

- `base` – WWDT peripheral base address
- `timeoutCount` – WWDT timeout value, count of WWDT clock tick.

`static inline void WWDT_SetWindowValue(WWDT_Type *base, uint32_t windowValue)`

Sets the WWDT window value.

The `WINDOW` register determines the highest `TV` value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in `WINDOW`, a watchdog event will occur. To disable windowing, set `windowValue` to `0xFFFFFFFF` (maximum possible timer value) so windowing is not in effect.

Parameters

- `base` – WWDT peripheral base address
- `windowValue` – WWDT window value.

`void WWDT_Refresh(WWDT_Type *base)`

Refreshes the WWDT timer.

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

- `base` – WWDT peripheral base address

`FSL_WWDT_DRIVER_VERSION`

Defines WWDT driver version.

`WWDT_FIRST_WORD_OF_REFRESH`

First word of refresh sequence

`WWDT_SECOND_WORD_OF_REFRESH`

Second word of refresh sequence

enum __wwdt_status_flags_t

WWDT status flags.

This structure contains the WWDT status flags for use in the WWDT functions.

Values:

enumerator kWWDT_TimeoutFlag

Time-out flag, set when the timer times out

enumerator kWWDT_WarningFlag

Warning interrupt flag, set when timer is below the value WDWARNINT

typedef struct __wwdt_config wwdt_config_t

Describes WWDT configuration structure.

struct __wwdt_config

#include <fsl_wwdt.h> Describes WWDT configuration structure.

Public Members

bool enableWwdt

Enables or disables WWDT

bool enableWatchdogReset

true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset

bool enableWatchdogProtect

true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time

uint32_t windowValue

Window value, set this to 0xFFFFFFFF if windowing is not in effect

uint32_t timeoutValue

Timeout value

uint32_t warningValue

Watchdog time counter value that will generate a warning interrupt. Set this to 0 for no warning

uint32_t clockFreq_Hz

Watchdog clock source frequency.

Chapter 3

Middleware

3.1 Wireless

3.1.1 NXP Wireless Framework and Stacks

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

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

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

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

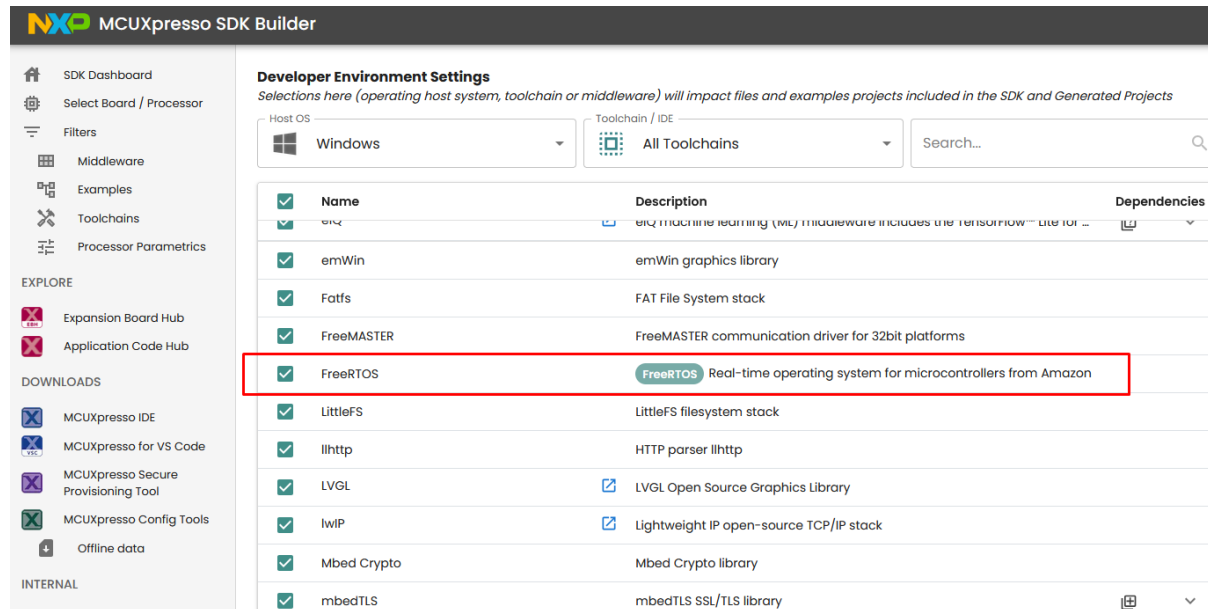
The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

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

List of examples The list of `freertos_examples`, their description and availability for individual supported MCUXpresso SDK development boards can be obtained here: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos_examples/index.html

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

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



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

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

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

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

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

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

FreeRTOS kernel for MCUXpresso SDK ChangeLog

Changelog FreeRTOS kernel for MCUXpresso SDK All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[Unreleased]**Added**

- Kconfig added CONFIG_FREERTOS_USE_CUSTOM_CONFIG_FRAGMENT config to optionally include custom FreeRTOSConfig fragment include file FreeRTOSConfig_frag.h. File must be provided by application.
- Added missing Kconfig option for configUSE_PICOLIBC_TLS.
- Add correct header files to build when configUSE_NEWLIB_REENTRANT and configUSE_PICOLIBC_TLS is selected in config.

[11.1.0_rev0]

- update amazon freertos version

[11.0.1_rev0]

- update amazon freertos version

[10.5.1_rev0]

- update amazon freertos version

[10.4.3_rev1]

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

[10.4.3_rev0]

- update amazon freertos version.

[10.4.3_rev0]

- update amazon freertos version.

[9.0.0_rev3]

- New features:
 - Tickless idle mode support for Cortex-A7. Add fsl_tickless_epit.c and fsl_tickless_generic.h in portable/IAR/ARM_CA9 folder.
 - Enabled float context saving in IAR for Cortex-A7. Added configUSE_TASK_FPU_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM_CA9 folder.
- Other changes:
 - Transformed ARM_CM core specific tickless low power support into generic form under freertos/Source/portable/low_power_tickless/.

[9.0.0_rev2]

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

[9.0.0_rev1]

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

[9.0.0_rev0]

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

[8.2.3]

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

FreeRTOS kernel Readme

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

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



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

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

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

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

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

To consume FreeRTOS-Kernel

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

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

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

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

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

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

– include/FreeRTOSConfig.h

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
  INTERFACE
    include
)

target_compile_definitions(freertos_config
  INTERFACE
    projCOVERAGE_TEST=0
)
```

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

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

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

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

FetchContent_MakeAvailable(freertos_kernel)
```

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

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

Consuming stand-alone - Cloning this repository

To clone using HTTPS:

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

Using SSH:

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

Repository structure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

return dnsStatus;
}

```

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

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

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

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

gcc can also produce an output file to be linked:

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

Building unit tests

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

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

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

Platform Prerequisites

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

Steps to build Unit Tests

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

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

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

MCUXpresso SDK: coreHTTP Client Library

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

coreHTTP Client Library

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

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

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

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

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

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

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

The HTTP client library can be built by either:

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

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

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

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

Building Unit Tests

Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

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

4.1.5 corejson

JSON parser.

Readme

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

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

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

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

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

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

Reference example

```

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

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

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

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

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

    return 0;
}

```

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

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

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

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

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

gcc can also produce an output file to be linked:

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

Building unit tests

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

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

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

Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

MCUXpresso SDK: coreMQTT Library

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

coreMQTT Client Library

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

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

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

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

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

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

Thus, the MQTT library can be built by either:

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

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

Format

The format of the username string with metrics is:

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

Where

- <Actual_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.
- <OS_Name> is the Operating System the application is running on (e.g. FreeRTOS)
- <OS_Version> is the version number of the Operating System (e.g. V10.4.3)
- <Hardware_Platform> is the Hardware Platform the application is running on (e.g. WinSim)
- <MQTT_Library_name> is the MQTT Client library being used (e.g. coreMQTT)
- <MQTT_Library_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual_Username = "iotuser", OS_Name = FreeRTOS, OS_Version = V10.4.3, Hardware_Platform_Name = WinSim, MQTT_Library_Name = coremqtt, MQTT_Library_version = 2.1.1. If username is not used, then "iotuser" can be removed.

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

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

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

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

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

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

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

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

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

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

Building Unit Tests

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

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

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

Platform Prerequisites

- Docker

or the following:

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

Steps to build Unit Tests

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

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

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

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

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

Platform	Location	Transport Interface Implementation
POSIX	AWS IoT Device SDK for Embedded C	POSIX sockets for TCP/IP and OpenSSL for TLS stack
FreeRTOS	FreeRTOS/FreeRTOS	FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack
FreeRTOS	FreeRTOS AWS Reference Integrations	Based on Secure Sockets Abstraction

Documentation

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

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

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

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

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

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

4.1.7 coremqtt-agent

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

Readme

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

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

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

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

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

To clone using HTTPS:

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

Using SSH:

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

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

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

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

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

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

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

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

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

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

Additional Considerations

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

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

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

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

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

Building Unit Tests

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

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

Unit Test Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

Location
coreMQTT Agent Demos
FreeRTOS/FreeRTOS

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

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

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

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

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

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

4.1.8 corepkcs11

PKCS #11 key management library.

Readme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Build Prerequisites

Library Usage For building the library the following are required:

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

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

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

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

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

Coverage Measurement and Instrumentation The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.
- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.
- **gcov** binary corresponding to your chosen compiler
- **lcov** from the [Linux Test Project](#)
- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.
2. Run **cmake** to construct a build tree: `cmake -S test -B build`
 - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.
 - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.
3. Build the test binaries: `cmake --build ./build --target all`
4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.
5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

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

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

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

Reference examples The FreeRTOS-Labs repository contains demos using the PKCS #11 library [here](#) using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

Porting Guide Documentation for porting corePKCS11 to a new platform can be found on the AWS [docs](#) web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

Related Example Implementations These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's [Platform Security Architecture](#).
- Microchip's [cryptoauthlib](#).
- Infineon's [Optiga Trust X](#).

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

Note that the latest included version of corePKCS11 may differ across repositories.

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

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

Security See *CONTRIBUTING* for more information.

License This library is licensed under the MIT-0 License. See the LICENSE file.

4.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme

MCUXpresso SDK: FreeRTOS-Plus-TCP Library This repository is a fork of FreeRTOS-Plus-TCP library (<https://github.com/FreeRTOS/freertos-plus-tcp>)(4.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

Introduction This branch contains unified IPv4 and IPv6 functionalities. Refer to the Getting started Guide (found [here](#)) for more details.

FreeRTOS-Plus-TCP Library FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the [MISRA coding standard](#). Any deviations from the MISRA C:2012 guidelines are documented under [MISRA Deviations](#). The library is validated for memory safety and data structure invariance through the [CBMC automated reasoning tool](#) for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

Getting started The easiest way to use the 4.0.0 version of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found [here](#)) Another way is to start with the pre-configured demo application project (found in [this directory](#)). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the [Documentation](#), and [API Reference](#).

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#). Please also refer to [FAQ](#) for frequently asked questions.

Also see the [Submitting a bugs/feature request](#) section of CONTRIBUTING.md for more details.

Note: All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

Upgrading to V3.0.0 and V3.1.0 In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a `source` directory. This change requires modification of any existing project(s) to include the modified source files and directories. There are examples on how to use the new files and directory structure. For an example based on the Xilinx Zynq-7000, use the code in this [branch](#) and follow these [instructions](#) to build and run the demo.

FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 20210.00 LTS release.

Generating pre V3.0.0 folder structure for backward compatibility: If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to [this](#).

Note: After running the script, while the `.c` files will have same names as the pre V3.0.0 source, the files in the `include` directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from [here](#).

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing `python --version`.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory that was created using the *Cloning this repository* step above. And then run `python <Path/to/the/script>/GenerateOriginalFiles.py`.

To consume FreeRTOS+TCP

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

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

```
FetchContent_Declare( freertos_plus_tcp
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
  GIT_TAG        master #Note: Best practice to use specific git-hash or tagged version
  GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

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

```

set( FREERTOS_PLUS_FAT_DEV_SUPPORT OFF CACHE BOOL "" FORCE)
# Select the native compile PORT
set( FREERTOS_PLUS_FAT_PORT "POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  # Eg. Zynq 2019_3 version of port
  set(FREERTOS_PLUS_FAT_PORT "ZYNQ_2019_3" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)

```

Consuming stand-alone This repository uses [Git Submodules](#) to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```

git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Using SSH:

```

git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Porting The porting guide is available on [this page](#).

Repository structure This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
 - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
 - This directory contains all the tests (unit tests and CBMC) and the dependencies ([FreeRTOS-Kernel/Litani-port](#)) the tests require.
- source/portable
 - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
 - The include directory has all the ‘core’ header files of FreeRTOS-Plus-TCP source.
- source
 - This directory contains all the [.c] source files.

Note At this time it is recommended to use `BufferAllocation_2.c` in which case it is essential to use the `heap_4.c` memory allocation scheme. See [memory management](#).

Kernel sources The FreeRTOS Kernel Source is in [FreeRTOS/FreeRTOS-Kernel repository](#), and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at `./test/FreeRTOS-Kernel` directory.

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

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

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