



MCUXpresso SDK Documentation

Release 25.12.00-pvw2



NXP
Nov 14, 2025



Table of contents

1	MIMXRT1010-EVK	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	31
1.3.1	Getting Started with MCUXpresso SDK Repository	31
1.4	Release Notes	44
1.4.1	MCUXpresso SDK Release Notes	44
1.5	ChangeLog	48
1.5.1	MCUXpresso SDK Changelog	48
1.6	Driver API Reference Manual	128
1.7	Middleware Documentation	128
1.7.1	FreeMASTER	128
1.7.2	FreeRTOS	128
1.7.3	File systemFatfs	128
2	MIMXRT1011	129
2.1	ADC: 12-bit Analog to Digital Converter Driver	129
2.2	ADC_ETC: ADC External Trigger Control	137
2.3	AIPSTZ: AHB to IP Bridge	141
2.4	AOI: Crossbar AND/OR/INVERT Driver	143
2.5	CACHE: ARMV7-M7 CACHE Memory Controller	146
2.6	Clock Driver	149
2.7	DCDC: DCDC Converter	180
2.8	DCP: Data Co-Processor	189
2.9	DCP AES blocking driver	194
2.10	DCP HASH driver	196
2.11	DCP AES non-blocking driver	198
2.12	DMAMUX: Direct Memory Access Multiplexer Driver	199
2.13	eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	201
2.14	EWM: External Watchdog Monitor Driver	220
2.15	FlexIO: FlexIO Driver	223
2.16	FlexIO Driver	223
2.17	FlexIO eDMA I2S Driver	241
2.18	FlexIO eDMA SPI Driver	244
2.19	FlexIO eDMA UART Driver	248
2.20	FlexIO I2C Master Driver	251
2.21	FlexIO I2S Driver	259
2.22	FlexIO SPI Driver	270
2.23	FlexIO UART Driver	283
2.24	FLEXRAM: on-chip RAM manager	293
2.25	FLEXSPI: Flexible Serial Peripheral Interface Driver	301
2.26	FLEXSPI eDMA Driver	317
2.27	GPC: General Power Controller Driver	320
2.28	GPIO: General-Purpose Input/Output Driver	322
2.29	GPT: General Purpose Timer	326

2.30	IOMUXC: IOMUX Controller	333
2.31	KPP: KeyPad Port Driver	345
2.32	Common Driver	347
2.33	Lin_lpuart_driver	360
2.34	LPI2C: Low Power Inter-Integrated Circuit Driver	367
2.35	LPI2C Master Driver	369
2.36	LPI2C Master DMA Driver	383
2.37	LPI2C Slave Driver	385
2.38	LPSPi: Low Power Serial Peripheral Interface	396
2.39	LPSPi Peripheral driver	396
2.40	LPSPi eDMA Driver	417
2.41	LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver	424
2.42	LPUART Driver	424
2.43	LPUART eDMA Driver	443
2.44	OCOTP: On Chip One-Time Programmable controller.	446
2.45	OTFAD: On The Fly AES-128 Decryption Driver	449
2.46	PIT: Periodic Interrupt Timer	452
2.47	PMU: Power Management Unit	456
2.48	PWM: Pulse Width Modulator	463
2.49	RTWDOG: 32-bit Watchdog Timer	487
2.50	SAI: Serial Audio Interface	493
2.51	SAI Driver	493
2.52	SAI EDMA Driver	520
2.53	SNVS: Secure Non-Volatile Storage	526
2.54	Secure Non-Volatile Storage High-Power	526
2.55	Secure Non-Volatile Storage Low-Power	535
2.56	SPDIF: Sony/Philips Digital Interface	544
2.57	SPDIF eDMA Driver	558
2.58	SRC: System Reset Controller Driver	561
2.59	TEMPMON: Temperature Monitor Module	567
2.60	TRNG: True Random Number Generator	569
2.61	WDOG: Watchdog Timer Driver	573
2.62	XBARA: Inter-Peripheral Crossbar Switch	578
3	Middleware	581
3.1	Motor Control	581
3.1.1	FreeMASTER	581
4	RTOS	619
4.1	FreeRTOS	619
4.1.1	FreeRTOS kernel	619
4.1.2	FreeRTOS drivers	619
4.1.3	backoffalgorithm	619
4.1.4	corehttp	619
4.1.5	corejson	619
4.1.6	coremqtt	620
4.1.7	corepkcs11	620
4.1.8	freertos-plus-tcp	620

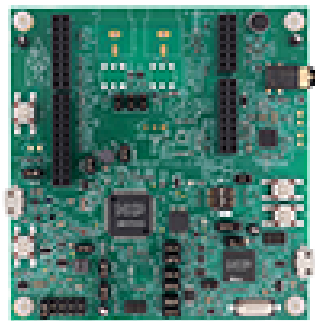
This documentation contains information specific to the evkmimxrt1010 board.

Chapter 1

MIMXRT1010-EVK

1.1 Overview

The NXP MIMXRT1010-EVK is a development board for the i.MX MIMXRT1011 500 MHz 32-bit ARM Cortex-M7 MCUs.



MCU device and part on board is shown below:

- Device: MIMXRT1011
- PartNumber: MIMXRT1011DAE5A

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

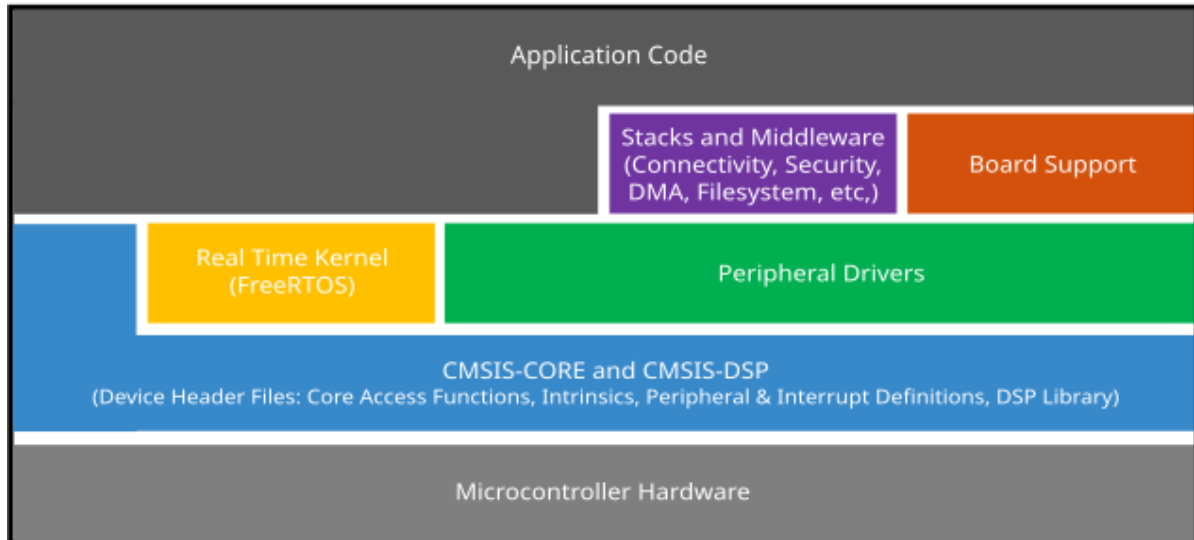
Overview

The NXP MCUXpresso software and tools offer comprehensive development solutions designed to optimize, ease and help accelerate embedded system development of applications based on general purpose, crossover and Bluetooth™-enabled MCUs from NXP. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case

examples to full demo applications. The MCUXpresso SDK contains optional RTOS integrations such as FreeRTOS and Azure RTOS, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for MIMXRT1010-EVK* (document MCUXSDKMIMXRT1011RN).

For more details about MCUXpresso SDK, see [MCUXpresso Software Development Kit \(SDK\)](#).



MCUXpresso SDK board support package folders

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

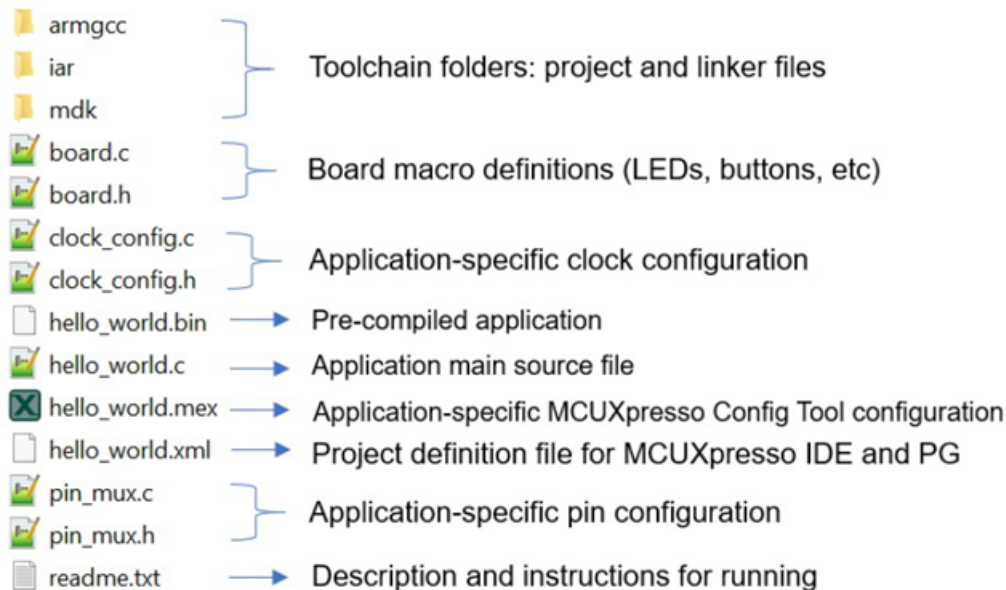
- `cmsis_driver_examples`: Simple applications intended to show how to use CMSIS drivers.
- `demo_apps`: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may leverage stacks and middleware.
- `driver_examples`: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

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

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

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

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



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

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

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

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

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

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

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

Run a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the MIMXRT1010-EVK hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

Build an example application The following steps guide you through opening the `hello_world` example application. These steps may change slightly for other example applications as some of these applications may have additional layers of folders in their path.

1. If not already done, open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

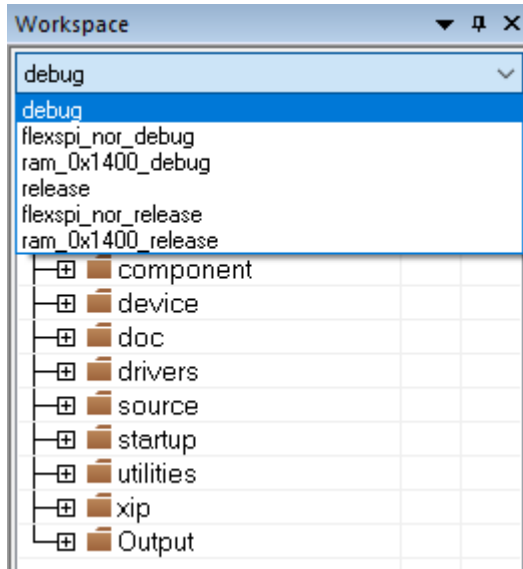
Using the MIMXRT1010-EVK hardware platform as an example, the `hello_world` workspace is located in

```
<install_dir>/boards/evkmimxrt1010/demo_apps/hello_world/iar/hello_world.eww
```

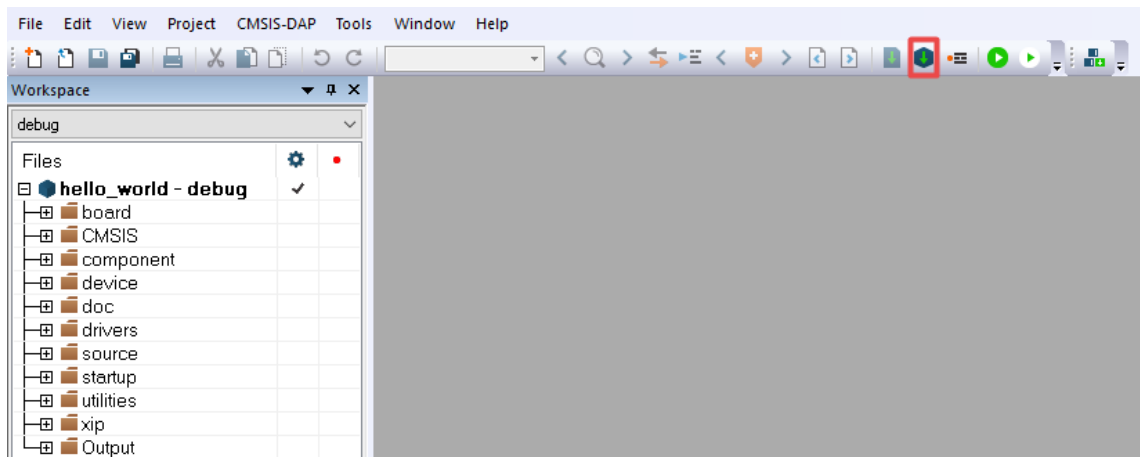
2. Select the desired build target from the drop-down.

There are six project configurations (build targets) supported for most MCUXpresso SDK projects:

- Debug – Compiler optimization is set to low, and debug information is generated for the executable. The linker file is RAM linker, where text and data section is put in internal TCM.
- Release – Compiler optimization is set to high, and debug information is not generated. The linker file is RAM linker, where text and data section is put in internal TCM.
- `ram_0x1400_debug` - Project configuration is same as debug target. The linker file is `RAM_0x1400` linker, where text is put in ITCM with offset `0x1400` and data section is put in DTCM.
- `ram_0x1400_release` - Project configuration is same as release target. The linker file is `RAM_0x1400` linker, where text is put in ITCM with offset `0x1400` and data section is put in DTCM.
- `flexspi_nor_debug` - Project configuration is same as Debug target. The linker file is `flexspi_nor` linker, where text is put in flash and data put in TCM.
- `flexspi_nor_release` - Project configuration is same as release target. The linker file is `flexspi_nor` linker, where text is put in flash and data put in TCM. For this example, select the “`hello_world – Debug`” target.



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

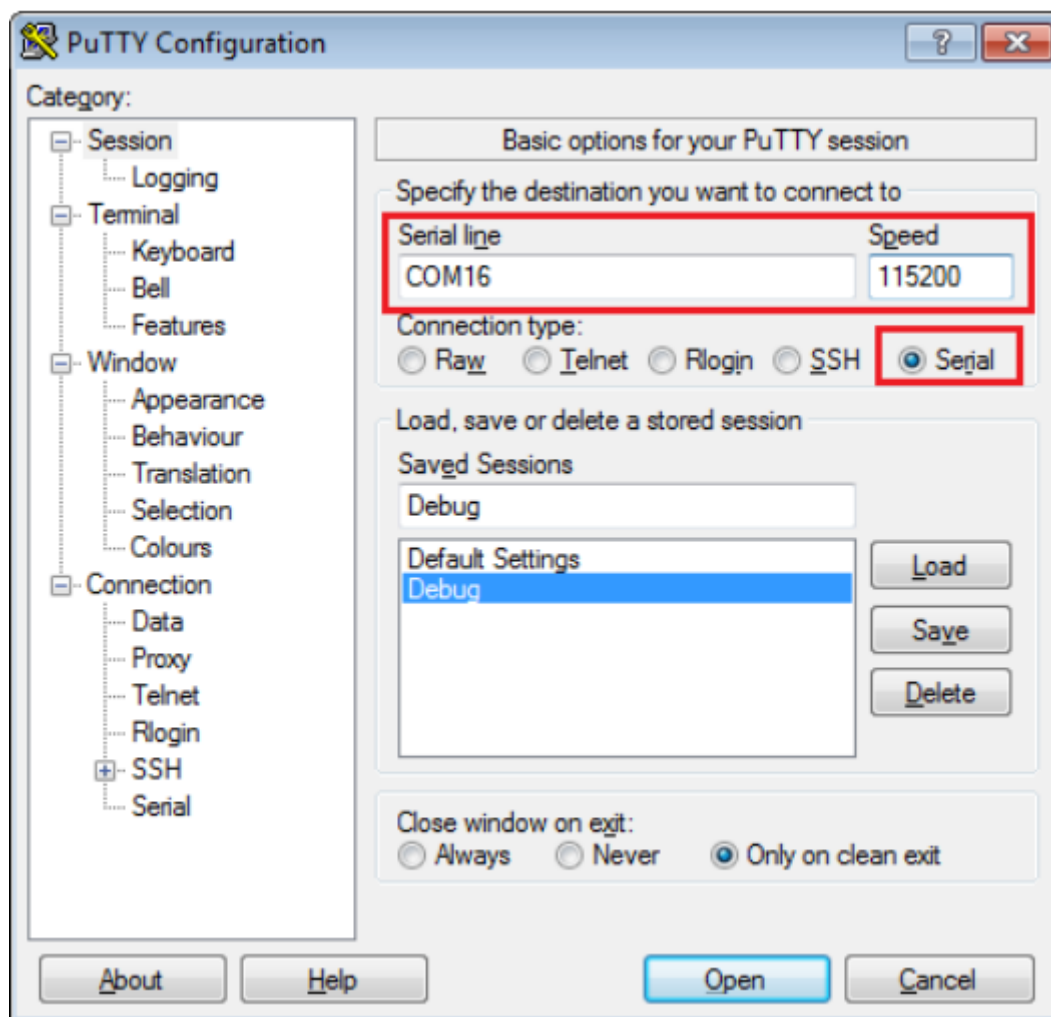


4. The build completes without errors.

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

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

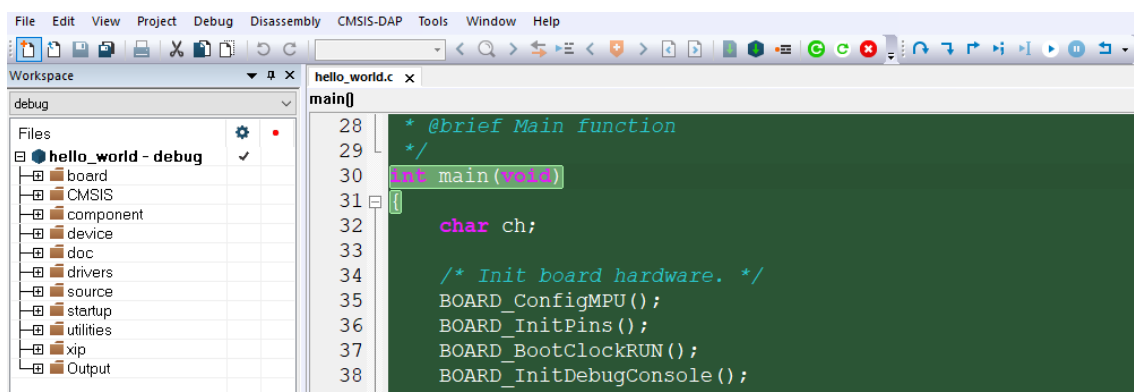
1. This board supports the CMSIS-DAP/mbd/DAPLink debug probe by default. Visit os.mbed.com/handbook/Windows-serial-configuration and follow the instructions to install the Windows® operating system serial driver. If running on Linux OS, this step is not required.
2. Connect the development platform to your PC via USB cable. Connect the USB cable to J41 and make sure SW8[1:4] is **0010b**.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits
 4. 1 stop bit



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



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

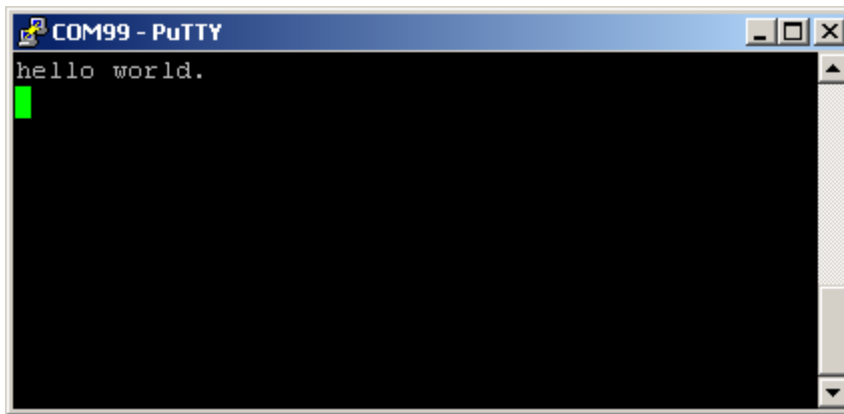


6. Run the code by clicking the **Go** button to start the application.



7. The `hello_world` application is now running and a banner is displayed on the terminal. If

this is not true, check your terminal settings and connections.



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

Run a demo using Keil® MDK/μVision

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

Install CMSIS device pack After the MDK tools are installed, Cortex® Microcontroller Software Interface Standard (CMSIS) device packs must be installed to fully support the device from a debug perspective. These packs include things such as memory map information, register definitions and flash programming algorithms. Follow these steps to install the MIMXRT1011 CMSIS pack.

1. Download the MIMXRT1011 packs from www.keil.com/dd2/pack/.
2. After downloading the DFP, double click to install it.

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

Build an example application

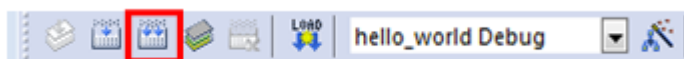
1. Open the desired example application workspace in:

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

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

```
<install_dir>/boards/evkmimxrt1010/demo_apps/hello_world/mdk/hello_world.uvmpw
```

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

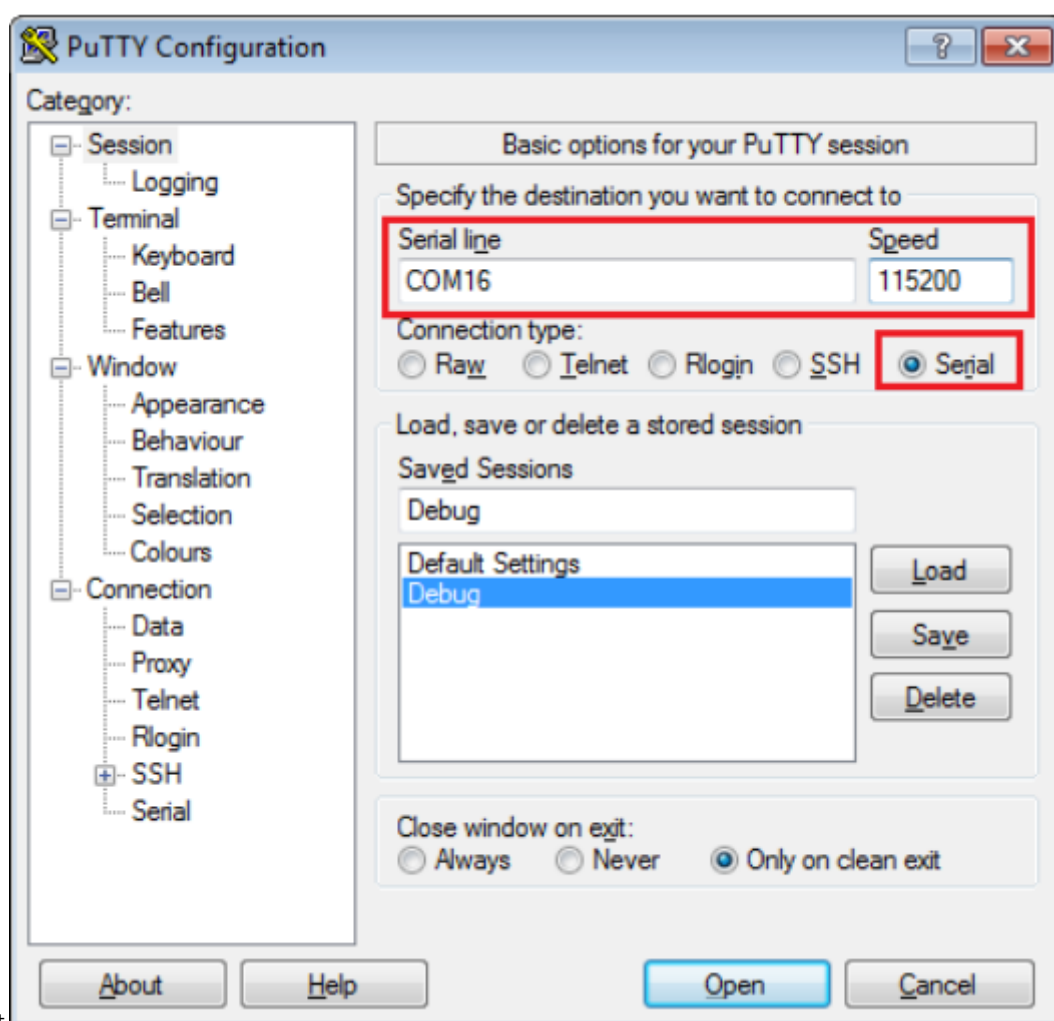


3. The build completes without errors.

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

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

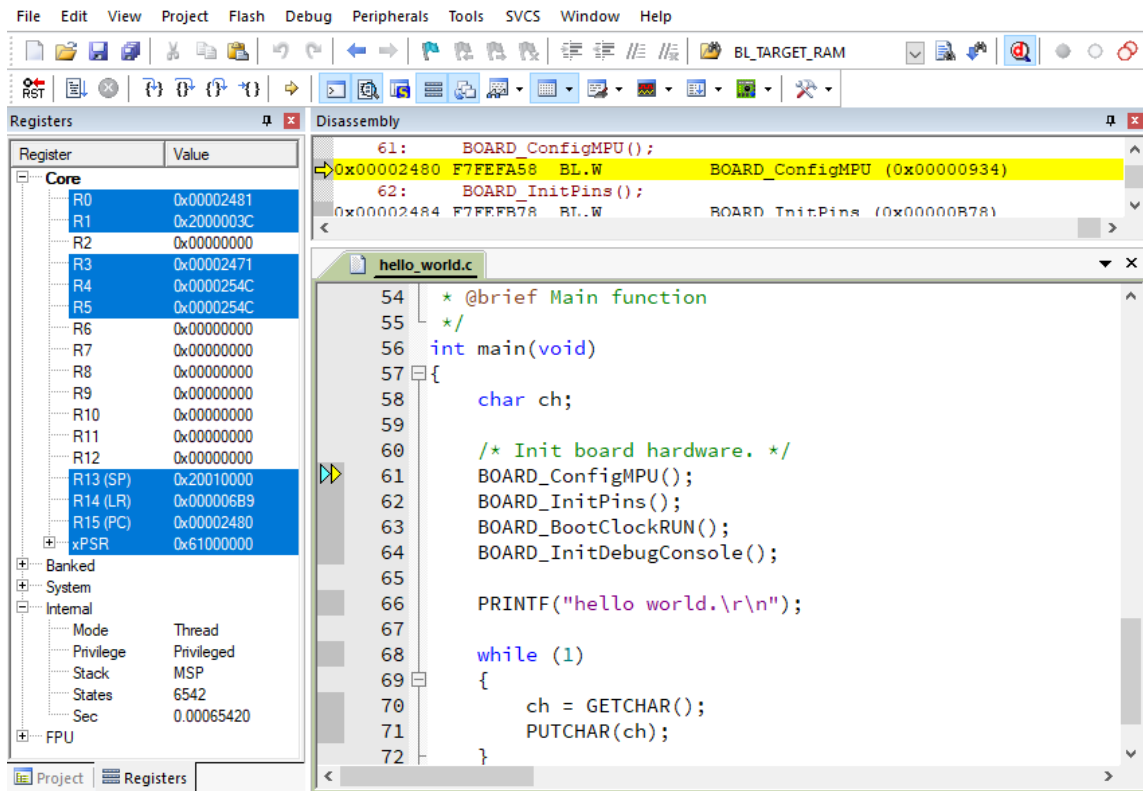
1. This board supports the CMSIS-DAP/mbed/DAPLink debug probe by default. Visit os.mbed.com/handbook/Windows-serial-configuration and follow the instructions to install the Windows® operating system serial driver. If running on Linux OS, this step is not required.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits



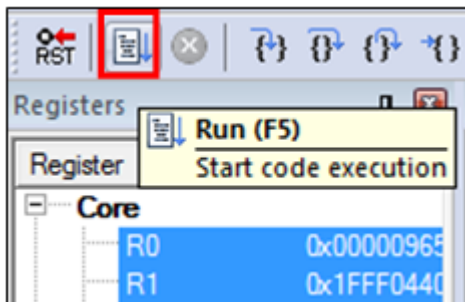
4. 1 stop bit

4. To debug the application, click **load** (or press the F8 key). Then, click the **Start/Stop Debug Session** button, highlighted in red in Figure 2. If using **J-Link** as the debugger, click **Project option > Debug > Settings > Debug > Port**, and select **SW**.

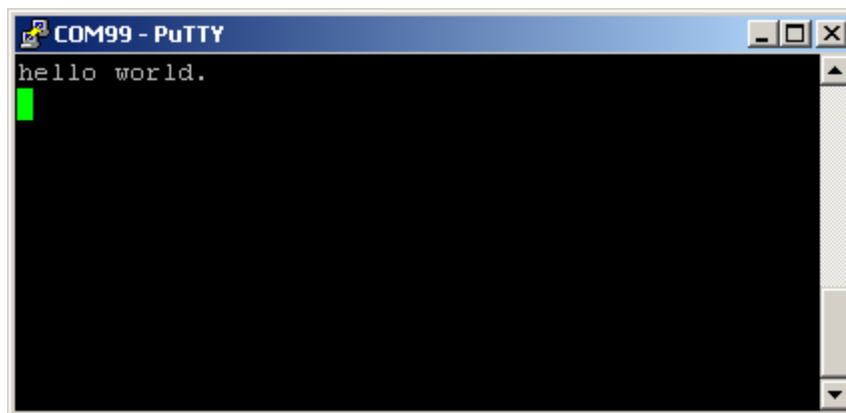
Note: When debugging with jlink, it expects one jlinkscript file named JLinkSettings.JLinkScript in the folder where the uVision project files are located. For details, see Segger Wiki. For the contents in this JLinkSettings.JLinkScript, use contents in evkmimxrt1020_sdram_init.jlinkscript.



5. Run the code by clicking **Run** to start the application, as shown in Figure 3.



The `hello_world` application is now running and a banner is displayed on the terminal, as shown in Figure 4. If this is not true, check your terminal settings and connections.



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

Run a demo using Arm® GCC

This section describes the steps to configure the command line Arm® GCC tools to build, run, and debug demo applications and necessary driver libraries provided in the MCUXpresso SDK. The `hello__world` demo application is targeted which is used as an example.

Note: GCC Arm Embedded 8.2.1 is used as an example in this document. The latest GCC version for this package is as described in the *MCUXpresso SDK Release Notes*.

Set up toolchain This section contains the steps to install the necessary components required to build and run an MCUXpresso SDK demo application with the Arm GCC toolchain, as supported by the MCUXpresso SDK. There are many ways to use Arm GCC tools, but this example focuses on a Windows operating system environment.

Install GCC Arm Embedded tool chain Download and run the installer from launchpad.net/gcc-arm-embedded. This is the actual toolset (in other words, compiler, linker, etc.). The GCC toolchain should correspond to the latest supported version, as described in *MCUXpresso SDK Release Notes Supporting MIMXRT1010-EVK* (document MCUXSDKMIMXRT101XRN).

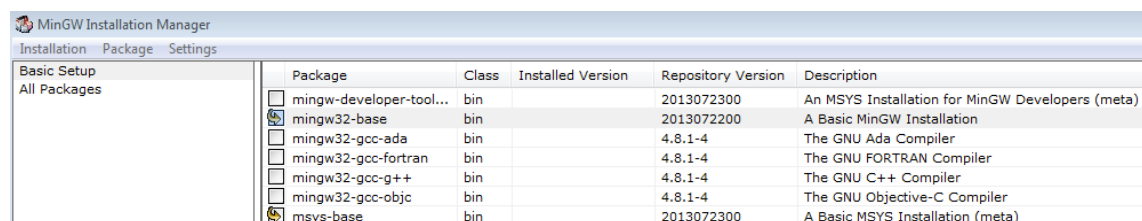
Parent topic:Set up toolchain

Install MinGW (only required on Windows OS) The Minimalist GNU for Windows (MinGW) development tools provide a set of tools that are not dependent on third-party C-Runtime DLLs (such as Cygwin). The build environment used by the MCUXpresso SDK does not use the MinGW build tools, but does leverage the base install of both MinGW and MSYS. MSYS provides a basic shell with a Unix-like interface and tools.

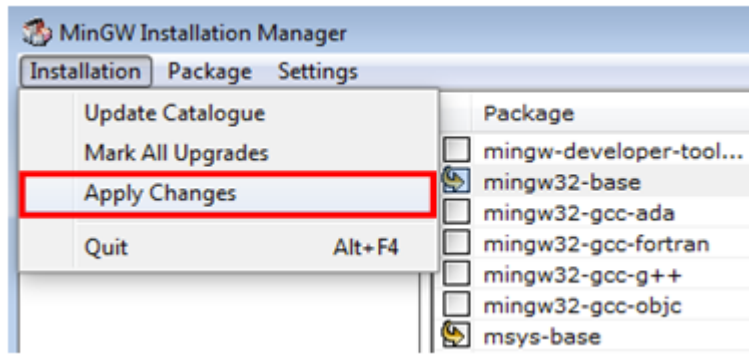
1. Download the latest MinGW mingw-get-setup installer from [MinGW](#).
2. Run the installer. The recommended installation path is `C:\MinGW`, however, you may install to any location.

Note: The installation path cannot contain any spaces.

3. Ensure that the **mingw32-base** and **msys-base** are selected under **Basic Setup**.



4. In the **Installation** menu, click **Apply Changes** and follow the remaining instructions to complete the installation.

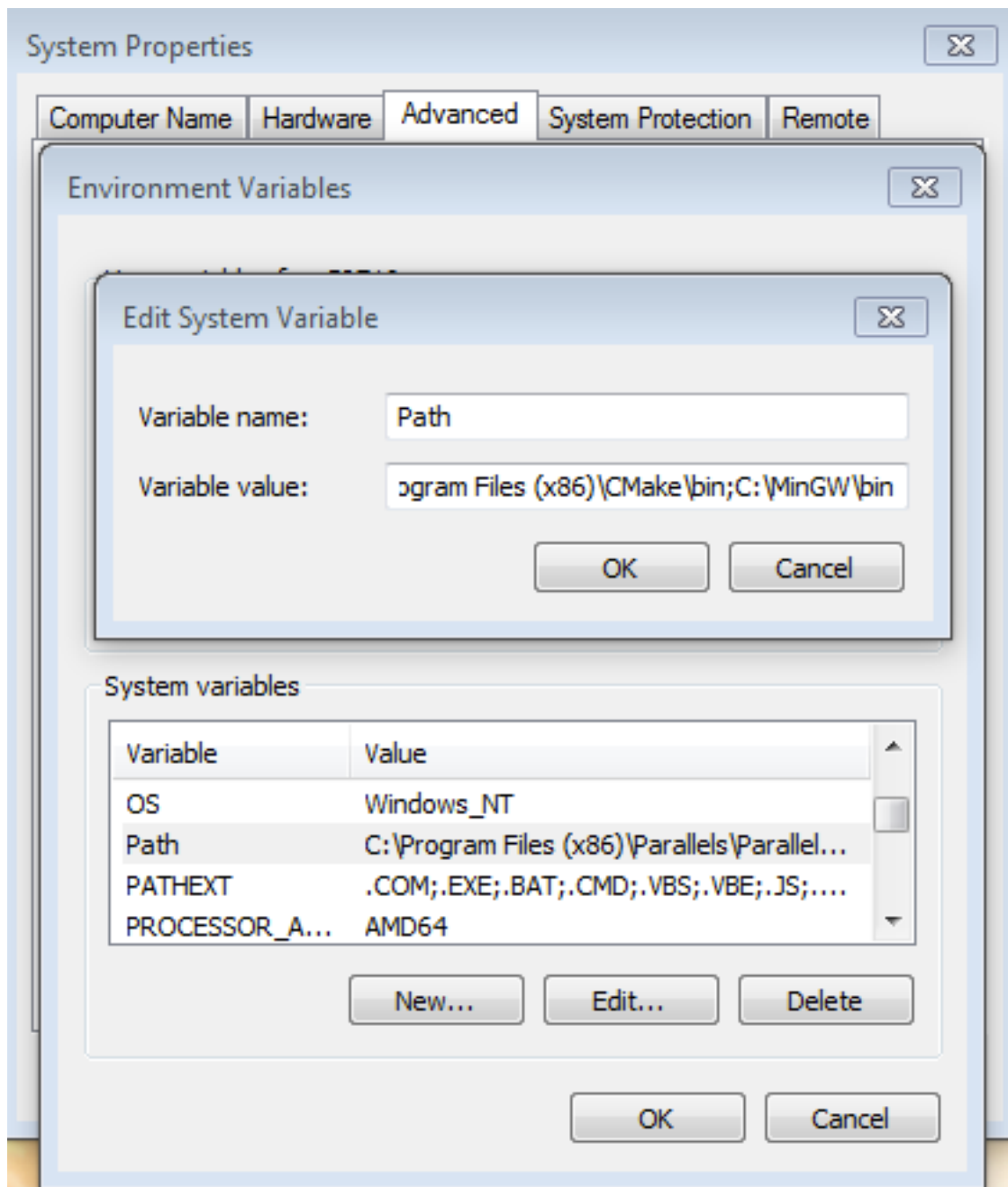


5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel->System and Security->System->Advanced System Settings** in the **Environment Variables...** section. The path is:

```
<mingw_install_dir>\bin
```

Assuming the default installation path, C:\MinGW, an example is shown below. If the path is not set correctly, the toolchain will not work.

Note: If you have C:\MinGW\msys\x.x\bin in your PATH variable (as required by Kinetis SDK 1.0.0), remove it to ensure that the new GCC build system works correctly.



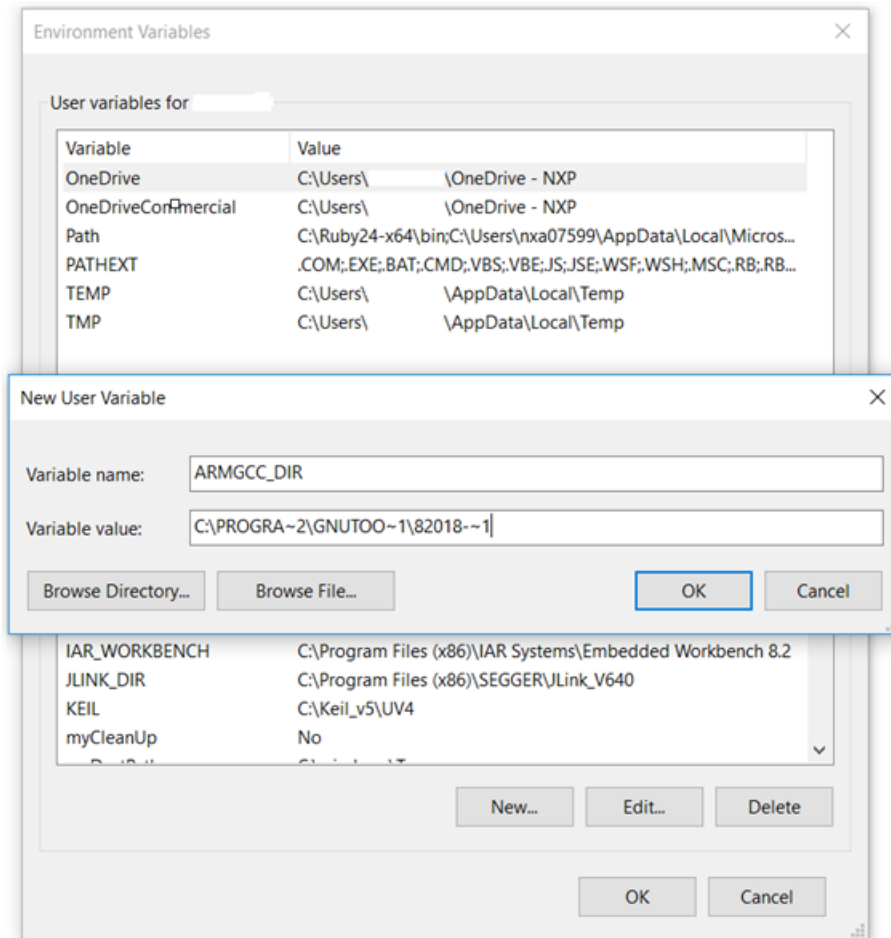
Parent topic:Set up toolchain

Add a new system environment variable for ARMGCC_DIR Create a new *system* environment variable and name it as `ARMGCC_DIR`. The value of this variable should point to the Arm GCC Embedded tool chain installation path. For this example, the path is:

See the installation folder of the GNU Arm GCC Embedded tools for the exact path name of your installation.

Short path should be used for path setting, you could convert the path to short path by running command for %I in (.) do echo %~sI in above path.

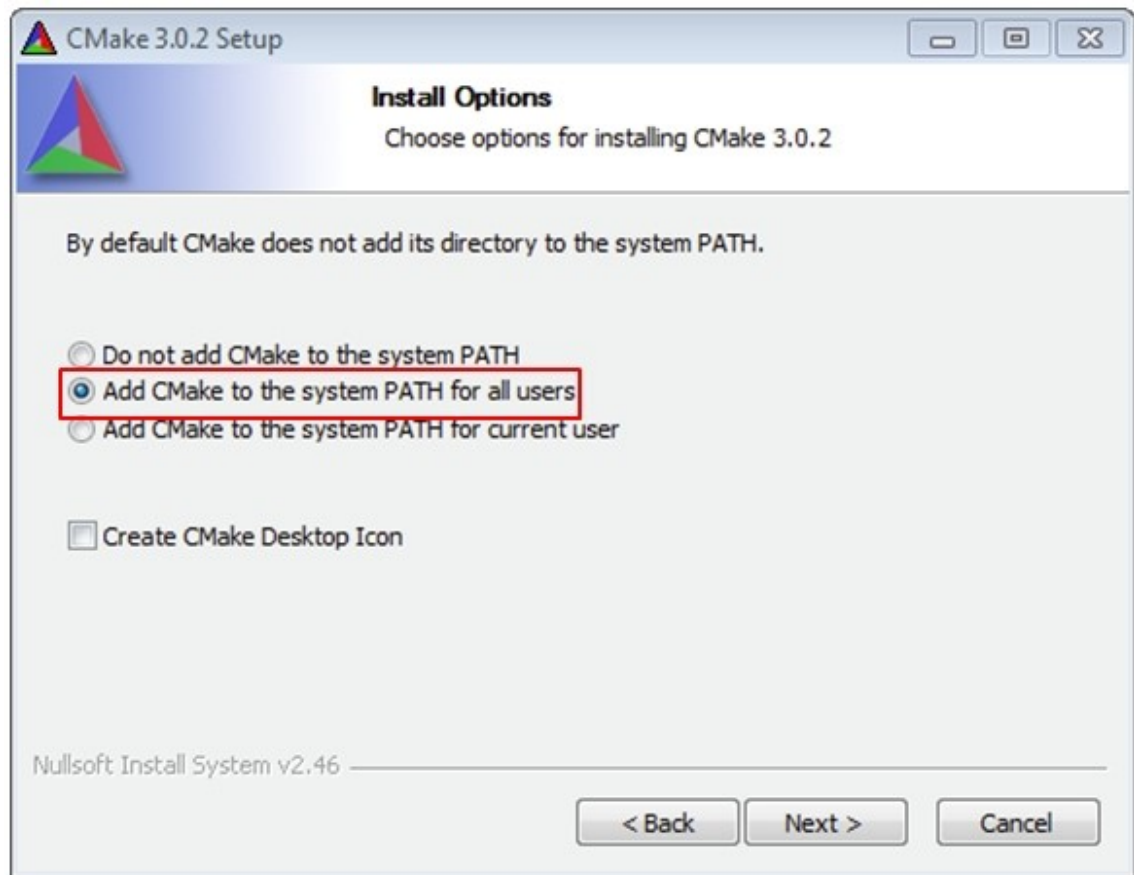
```
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>for %I in (.) do echo %~sI
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>echo C:\PROGRA~2\GNUTOO~1\82018~1
C:\PROGRA~2\GNUTOO~1\82018~1
```



Parent topic: Set up toolchain

Install CMake

1. Download CMake 3.0.x from www.cmake.org/cmake/resources/software.html.
2. Install CMake, ensuring that the option **Add CMake to system PATH** is selected when installing. The user chooses to select whether it is installed into the PATH for all users or just the current user. In this example, it is installed for all users.



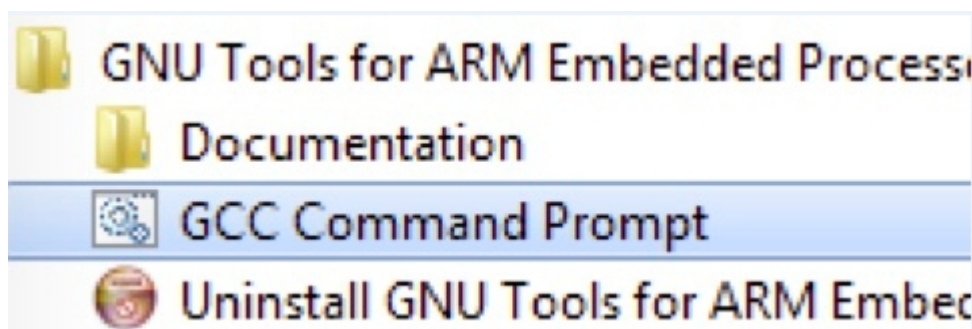
3. Follow the remaining instructions of the installer.
4. You may need to reboot your system for the PATH changes to take effect.
5. Make sure `sh.exe` is not in the Environment Variable PATH. This is a limitation of mingw32-make.

Parent topic:Set up toolchain

Parent topic:[Run a demo using Arm® GCC](#)

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

1. Open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system **Start** menu, go to **Programs > GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.



2. Change the directory to the example application project directory which has a path similar to the following:

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

For this example, the exact path is:

```
><install_dir>/examples/evkmimxrt1010/demo_apps/hello_world/armgcc
```

Note: To change directories, use the `cd` command.

3. Type **build_debug.bat** on the command line or double click on **build_debug.bat** file in Windows Explorer to build it. The output is as shown in Figure 2.

```
[ 95%] Building C object CMakeFiles/hello_world.elf.dir/C:/z_CMX/SDK/EVK-MIMXRT1010/boards/evkmimxrt1010/xip/evkmimxrt1010_flexspi_nor_config.c.obj
[100%] Linking C executable flexspi_nor_release\hello_world.elf
[100%] Built target hello_world.elf
C:\z_CMX\SDK\EVK-MIMXRT1010\boards\evkmimxrt1010\demo_apps\hello_world\armgcc>IF "" == "" (pause )
Press any key to continue . . .
```

Parent topic: [Run a demo using Arm® GCC](#)

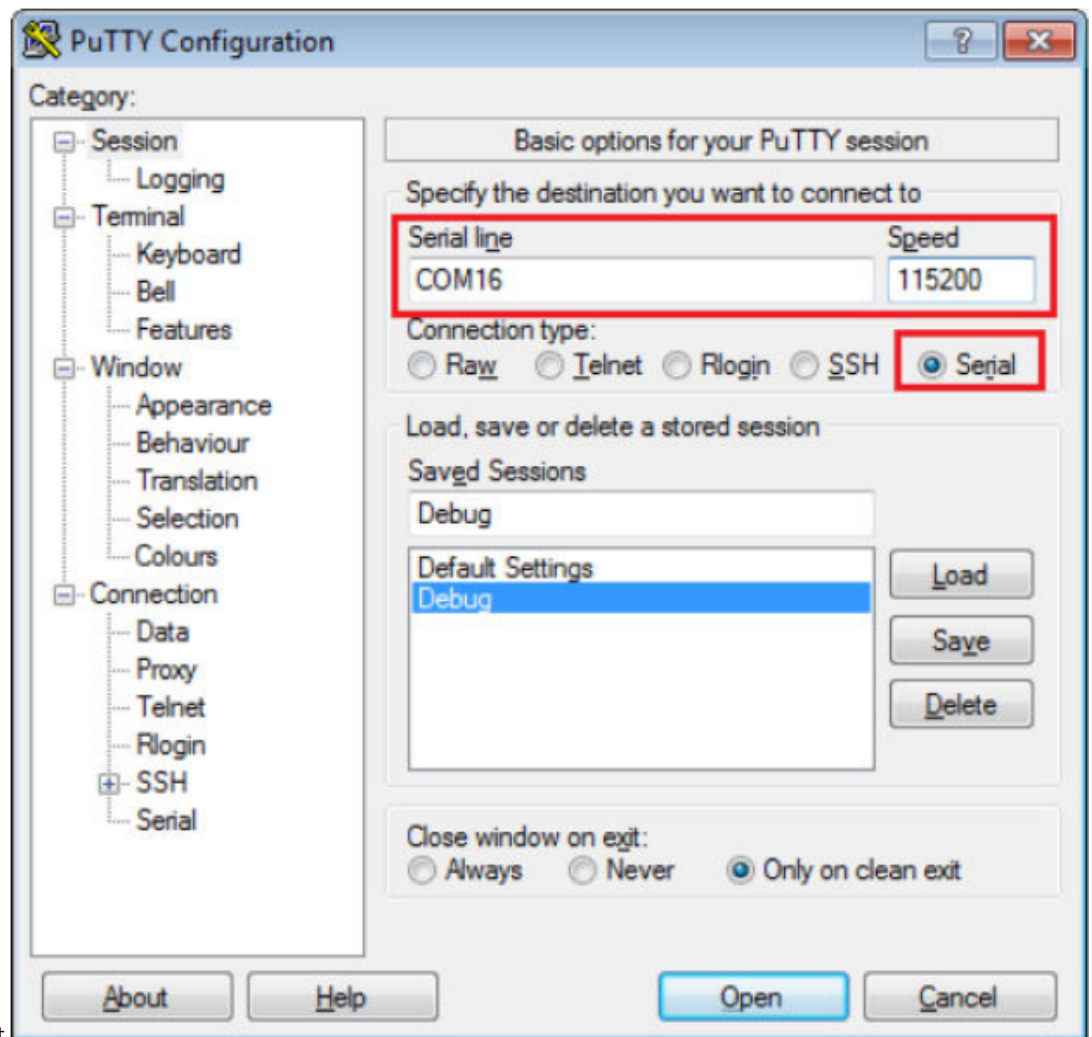
Run an example application This section describes steps to run a demo application using J-Link GDB Server application. To perform this exercise, make sure that either:

- The OpenSDA interface on your board is programmed with the J-Link OpenSDA firmware. If your board does not support OpenSDA, then a standalone J-Link pod is required.
- You have a standalone J-Link pod that is connected to the debug interface of your board.

Note: Some hardware platforms require hardware modification in order to function correctly with an external debug interface.

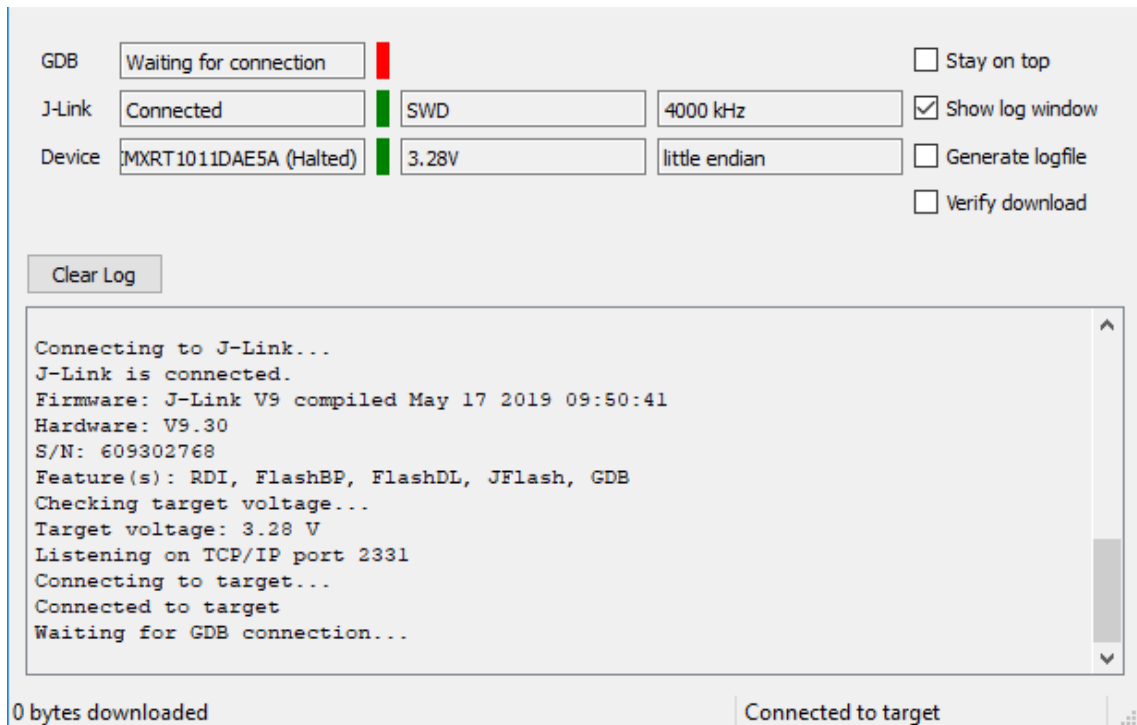
After the J-Link interface is configured and connected, follow these steps to download and run the demo applications:

1. This board supports the J-Link debug probe. Before using it, install SEGGER software, which can be downloaded from <http://www.segger.com>.
2. Connect the development platform to your PC via USB cable between the OpenSDA USB connector and the PC USB connector. If using a standalone J-Link debug pod, also connect it to the SWD/JTAG connector of the board.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see Appendix A). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits

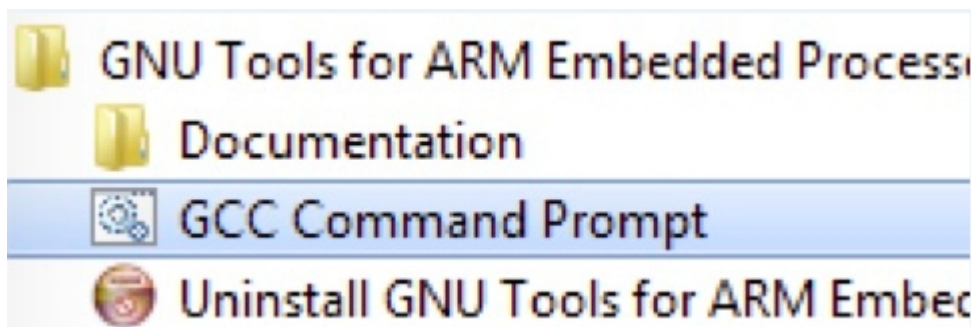


4. 1 stop bit

4. Open the J-Link GDB Server application. Go to the SEGGER install folder. For example, *C:\Program Files(x86)\SEGGER\JLink_Vxxx*. Open the command windows. For Debug and Release targets, use the *JLinkGDBServer.exe* command. For the
5. The target device selection chosen for this example is **MIMXRT1011DAE5A**.
6. After it is connected, the screen should resemble Figure 2.



7. If not already running, open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system **Start menu**, go to **Programs > GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.



8. Change to the directory that contains the example application output. The output can be found in using one of these paths, depending on the build target selected:

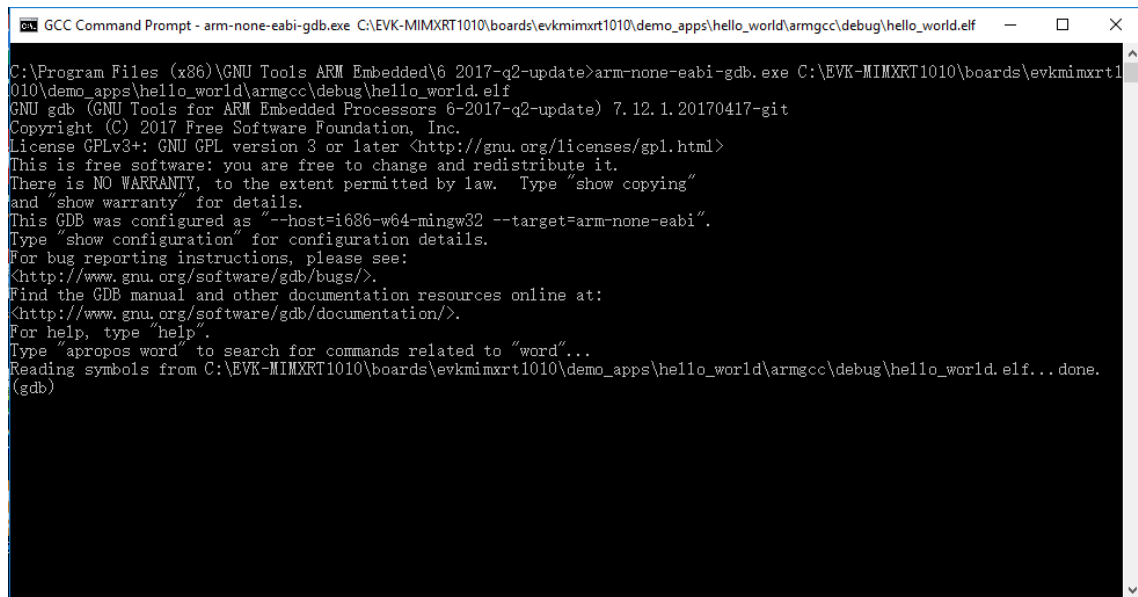
```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc/debug
```

```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc/release
```

For this example, the path is:

```
<install_dir>/boards/evkmimxrt1010/demo_apps/hello_world/armgcc/debug
```

9. Run the `arm-none-eabi-gdb.exe <application_name>.elf`. For this example, it is `arm-none-eabi-gdb.exe hello_world.elf`.



```

GCC Command Prompt - arm-none-eabi-gdb.exe C:\EVK-MIMXRT1010\boards\evkmimxrt1010\demo_apps\hello_world\armgcc\debug\hello_world.elf
C:\Program Files (x86)\GNU Tools ARM Embedded\6 2017-q2-update>arm-none-eabi-gdb.exe C:\EVK-MIMXRT1010\boards\evkmimxrt1010\demo_apps\hello_world\armgcc\debug\hello_world.elf
GNU gdb (GNU Tools for ARM Embedded Processors 6-2017-q2-update) 7.12.1.20170417-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from C:\EVK-MIMXRT1010\boards\evkmimxrt1010\demo_apps\hello_world\armgcc\debug\hello_world.elf...done.
(gdb)

```

10. Run these commands:

1. target remote localhost:2331
2. monitor reset
3. monitor halt
4. load

11. The application is now downloaded and halted at the reset vector. Execute the monitor go command to start the demo application.

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

Parent topic: [Run a demo using Arm® GCC](#)

Run a demo using MCUXpresso IDE

Note:

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

MCUXpresso IDE is not supported in this release.

This section describes the steps required to configure MCUXpresso IDE to build, run, and debug example applications. The hello_world demo application targeted for the MIMXRT1010-EVK platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

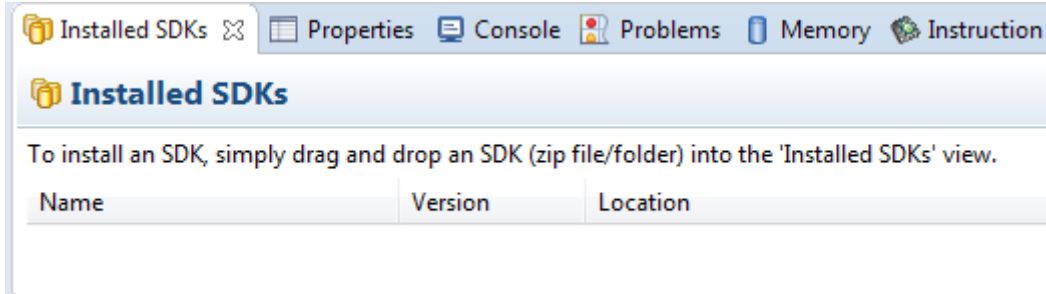
Note: By default, three macros, XIP_EXTERNAL_FLASH=1, XIP_BOOT_HEADER_ENABLE=1, and XIP_BOOT_HEADER_DCD_ENABLE=1, are set in the project. If you do not use Board_Flash in the project, these macros should be removed or set value to 0 in project settings.

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

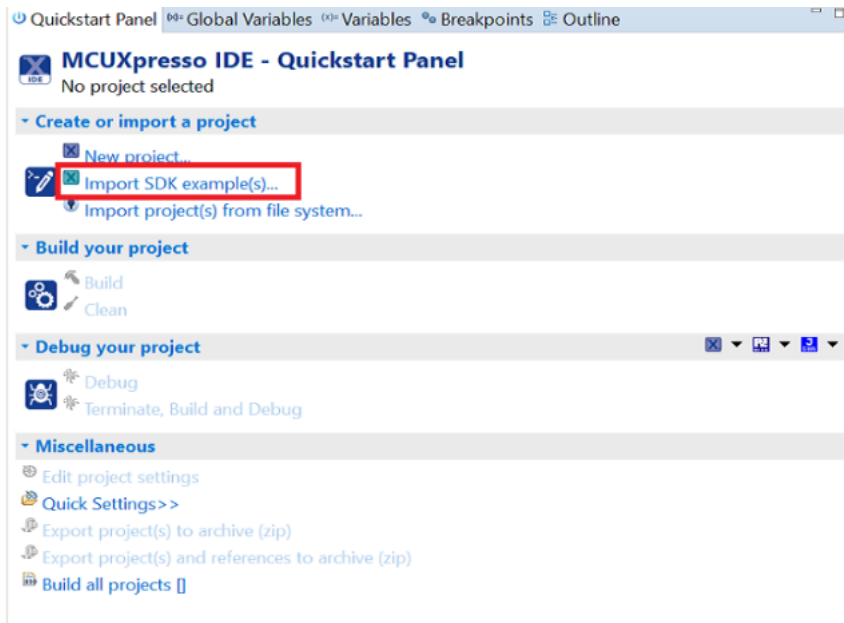
Parent topic: [Run a demo using MCUXpresso IDE](#)

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

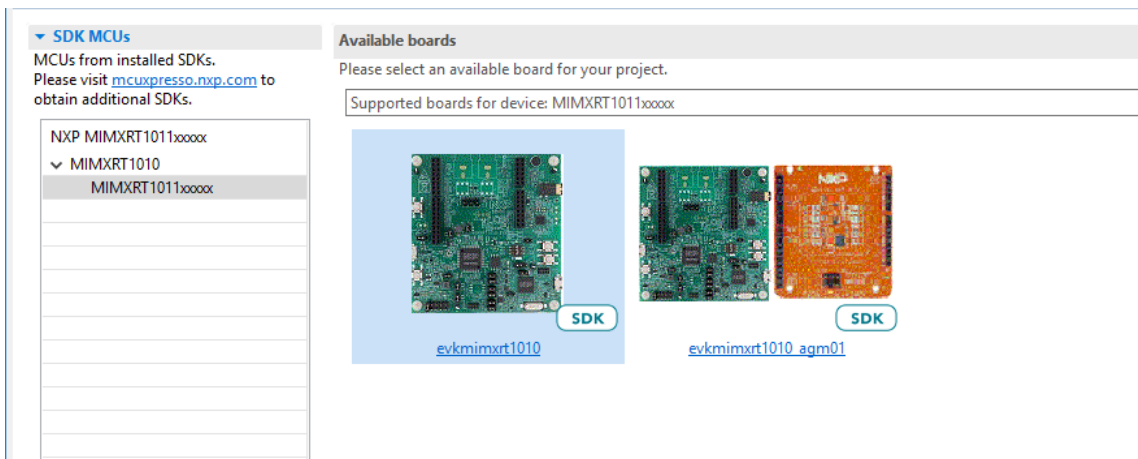
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install the MCUXpresso SDK. In the window that appears, click **OK** and wait until the import has finished.



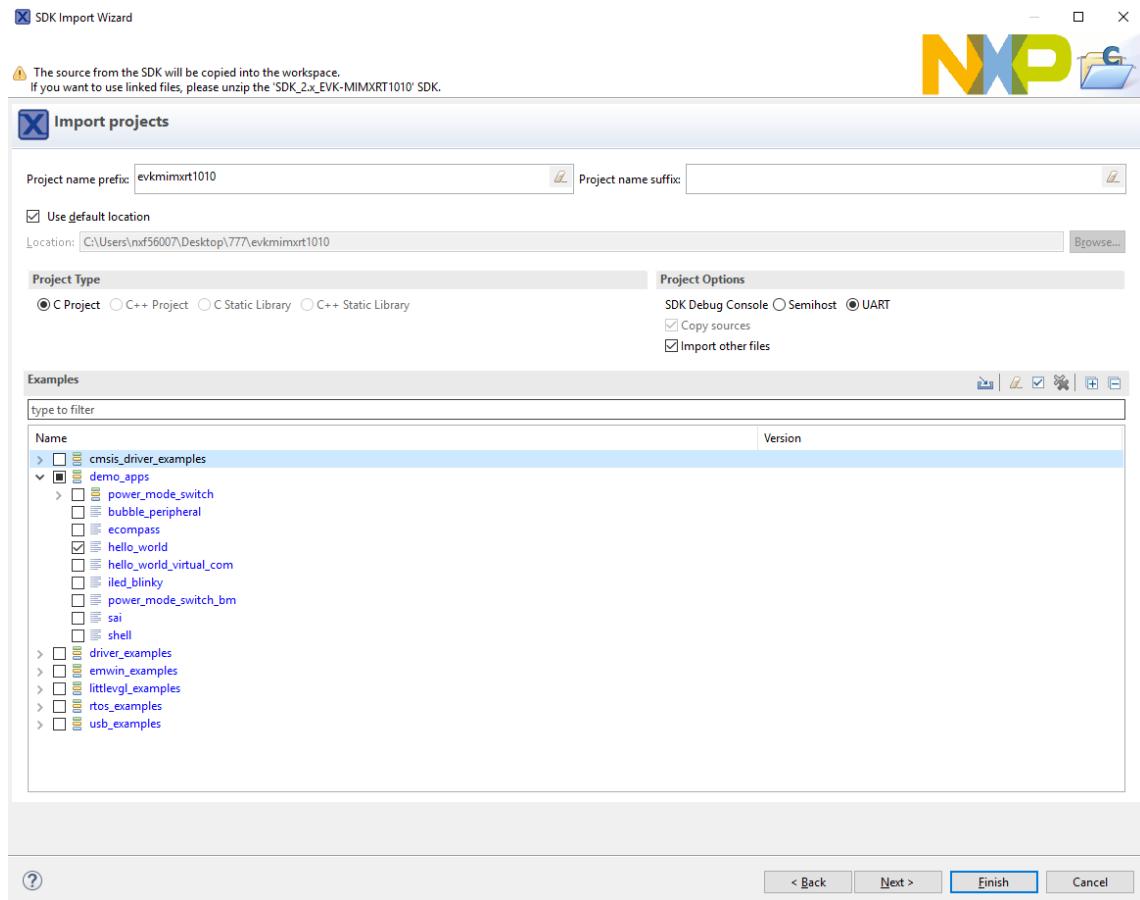
2. On the **Quickstart Panel**, click **Import SDK example(s)...**, as shown in Figure 2.



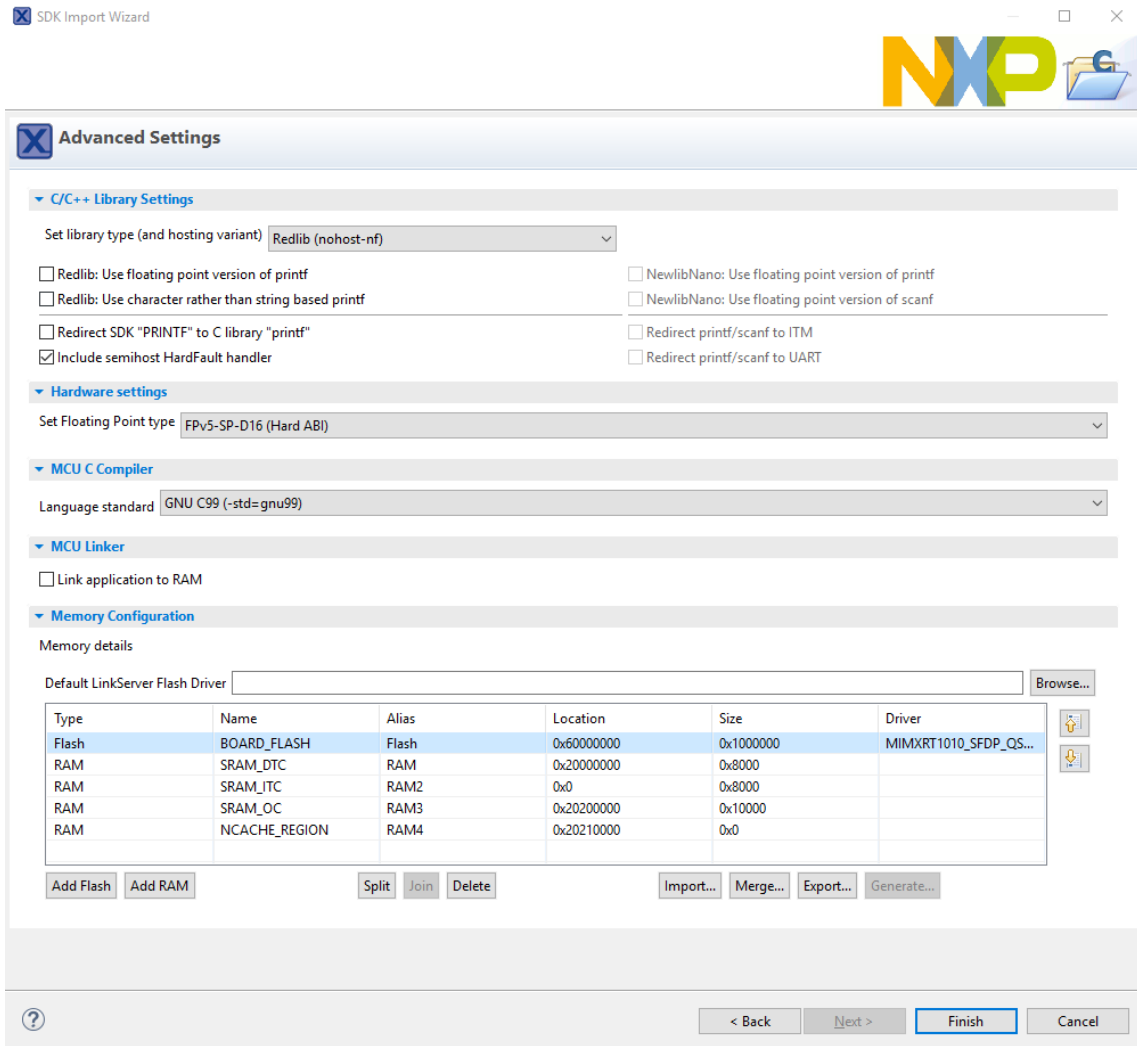
3. In the window that appears, expand the **MIMXRT1011** folder and select **MIMXRT1011xxxxx**. Then, select **evkmimxrt1010** and click **Next**, as shown in Figure 3.



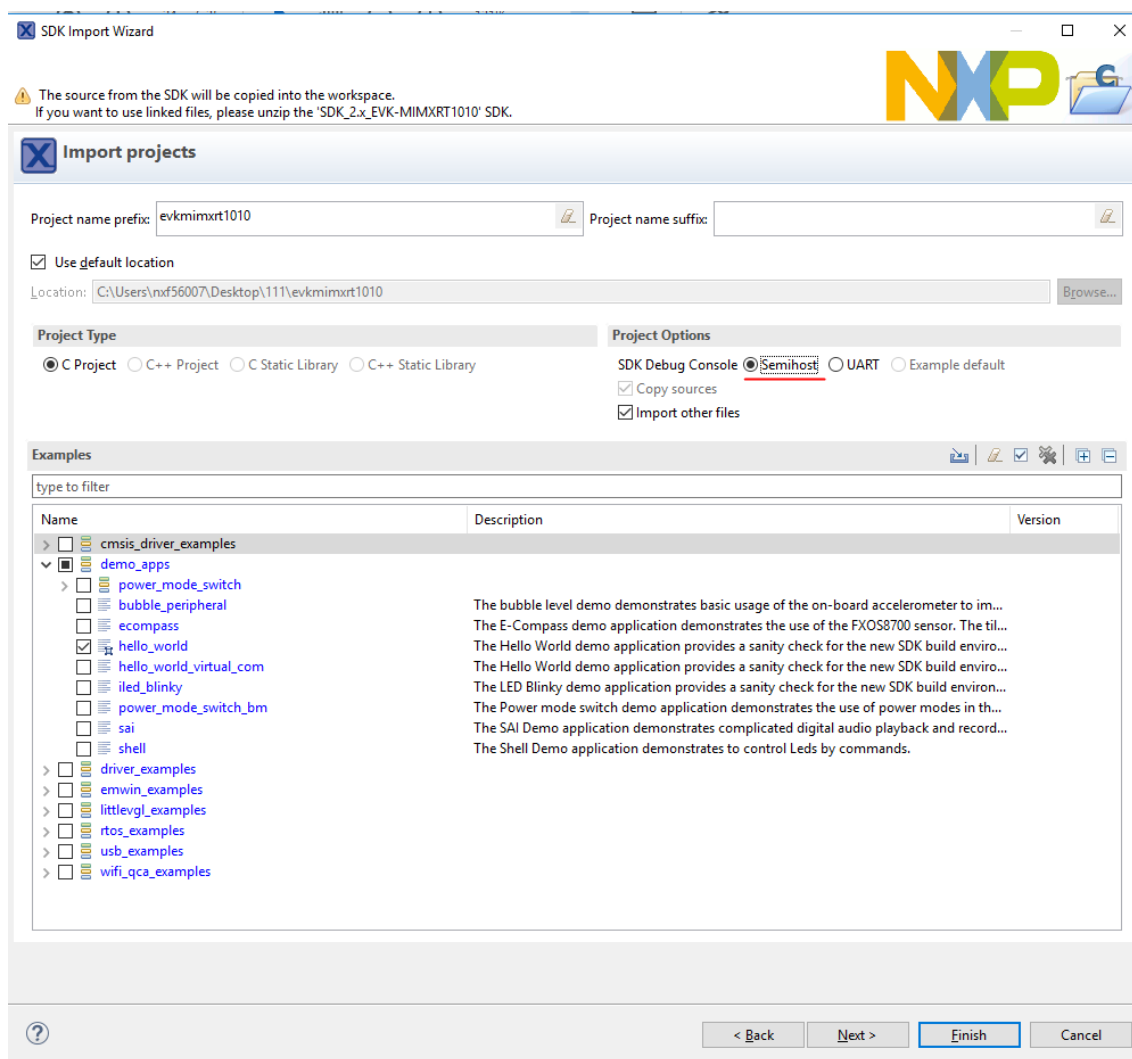
- Expand the `demo_apps` folder, select `hello_world`, and then click **Next**, as shown in *#SELECTIONTHELOWORLDCASE*.

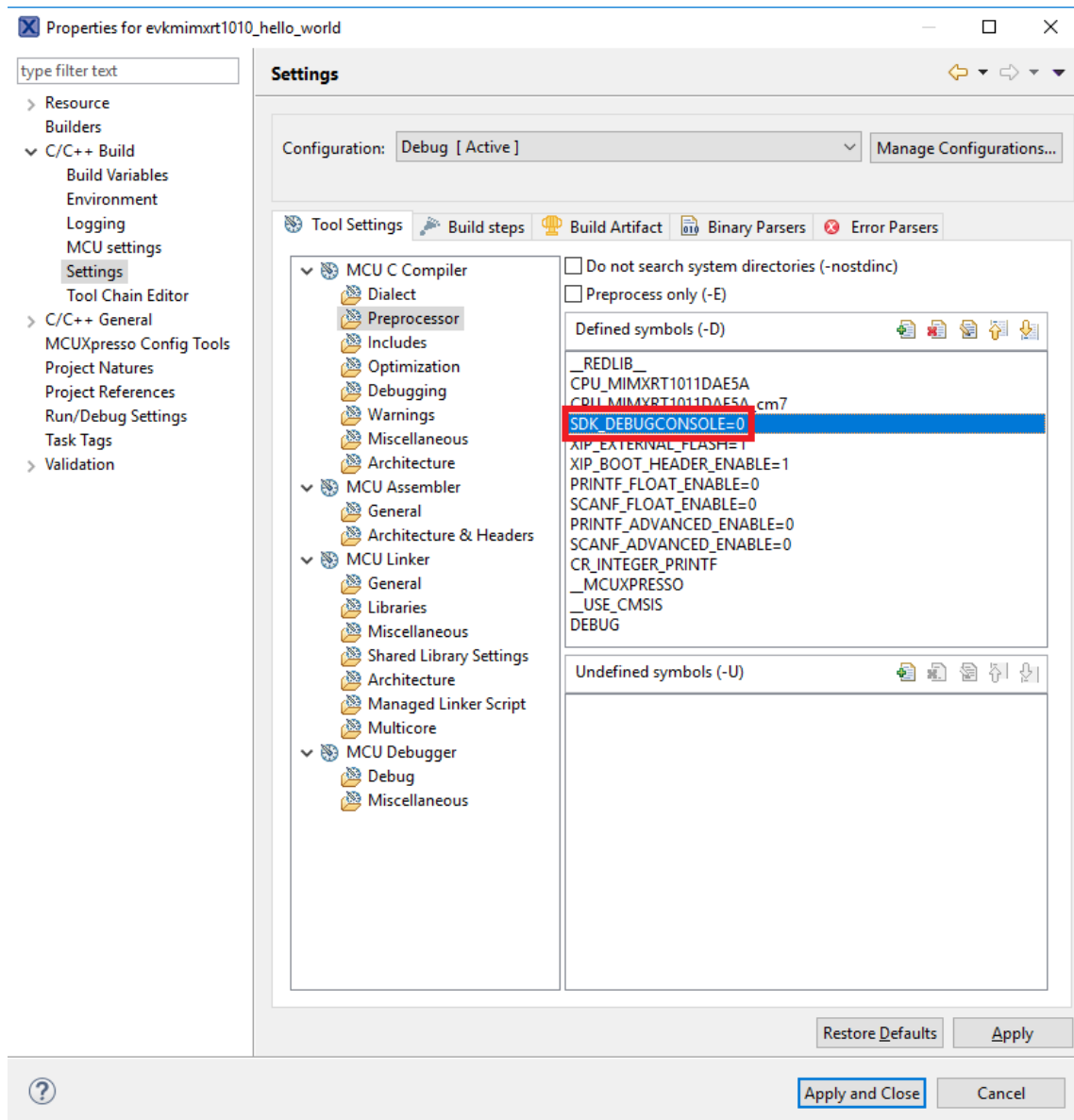


- Ensure the option **Redlib: Use floating point version of printf** is selected if the cases print floating point numbers on the terminal (for demo applications such as `dac32_adc12`, `dac_adc`, `dac_cadc`, `ecompass`, `sai`, `coremark`, `mbdctl_benchmark`, `wolfssl_benchmark`, and for `mmcau_examples` such as `mmcau_api`). Otherwise, there is no need to select it. Click **Finish**.

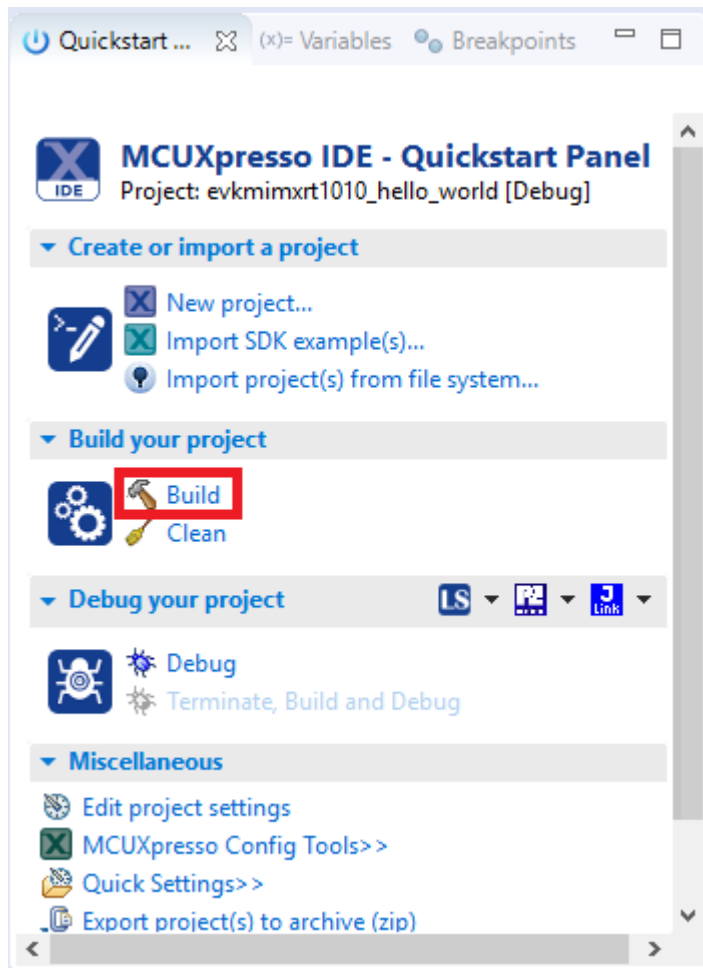


Note: If you want to use semihost to print log, first select the **Semihost** button when importing projects, as shown in Figure 6.





6. On the **Quickstart** panel, click **build** evkmimxrt1010_demo_apps_hello_world \[Debug\], as shown in Figure 8.



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

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

To download and run the application, perform these steps:

1. On the **Quickstart Panel**, click **Debug evkmimxrt1010_demo_apps_hello_world [Debug]**.
2. The first time you debug a project, the **Debug Emulator Selection Dialog** is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)
3. The application is downloaded to the target and automatically runs to `main()`.
4. Start the application by clicking the **Resume** button.



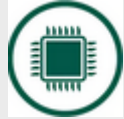
The `hello_world` application is now running and a banner is displayed on the MCUXpresso IDE console window. If this is not the case, check your terminal settings and connections.

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

MCUXpresso Config Tools

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

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

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

	Clock tool	For system clock configuration	
	Peripherals tools	For configuration of other peripherals	
	TEE tool	Configures access policies for memory area and peripherals helping to protect and isolate sensitive parts of the application.	
	Device Configuration tool	Configures Device Configuration Data (DCD) contained in the program image that the Boot ROM code interprets to set up various on-chip peripherals prior to the program launch.	

MCUXpresso Config Tools can be accessed in the following products:

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

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

MCUXpresso IDE New Project Wizard

MCUXpresso IDE features a new project wizard. The wizard provides functionality for the user to create new projects from the installed SDKs (and from pre-installed part support). It offers user the flexibility to select and change multiple builds. The wizard also includes a library and

provides source code options. The source code is organized as software components, categorized as drivers, utilities, and middleware.

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



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

How to determine COM port

This section describes the steps necessary to determine the debug COM port number of your NXP hardware development platform.

1. **Linux:** The serial port can be determined by running the following command after the USB Serial is connected to the host:

```
$ dmesg | grep "ttyUSB"
[503175.307873] usb 3-12: cp210x converter now attached to ttyUSB0
[503175.309372] usb 3-12: cp210x converter now attached to ttyUSB1
```

There are two ports, one is Cortex-A core debug console and the other is for Cortex M4.

2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click on the **Start** menu and type **Device Manager** in the search bar.
3. In the Device Manager, expand the **Ports (COM & LPT)** section to view the available ports. The COM port names will be different for all the NXP boards.

How to define IRQ handler in CPP files

With MCUXpresso SDK, users could define their own IRQ handler in application level to override the default IRQ handler. For example, to override the default PIT_IRQHandler define in startup_DEVICE.s, application code like app.c can be implement like:


```
c
void PIT_IRQHandler(void)
{
    // Your code
}
```

When application file is CPP file, like app.cpp, then `extern "C"` should be used to ensure the function prototype alignment.

```
cpp
extern "C" {
    void PIT_IRQHandler(void);
}
void PIT_IRQHandler(void)
{
    // Your code
}
```

How to add or remove boot header for XIP targets

The MCUXpresso SDK for i.MX RT1010 provides `flexspi_nor_debug` and `flexspi_nor_release` targets for each example and/or demo which supports XIP (eXecute-In-Place). These two targets add `XIP_BOOT_HEADER` to the image by default. Because of this, ROM can boot and run this image directly on external flash.

Macros for the boot leader:

- The following two macros are added in `flexspi_nor` targets to support XIP, as described in *how_to_add_or_remove_boot_header_for_xip_targets.md#TABLE_SUPPORTINGXIP*.

^

| **XIP_EXTERNAL_FLASH** | 1: Exclude the code which will change the clock of flexspi. | 0: Make no changes. | | **XIP_BOOT_HEADER_ENABLE** | 1: Add FLEXSPI configuration block, image vector table, and boot data to the image by default. | 0: Add nothing to the image by default. |

- how_to_add_or_remove_boot_header_for_xip_targets.md#TABLE_EFFECTSONBUILTIMAGE* shows the different effect on the built image with a different combination of these macros:

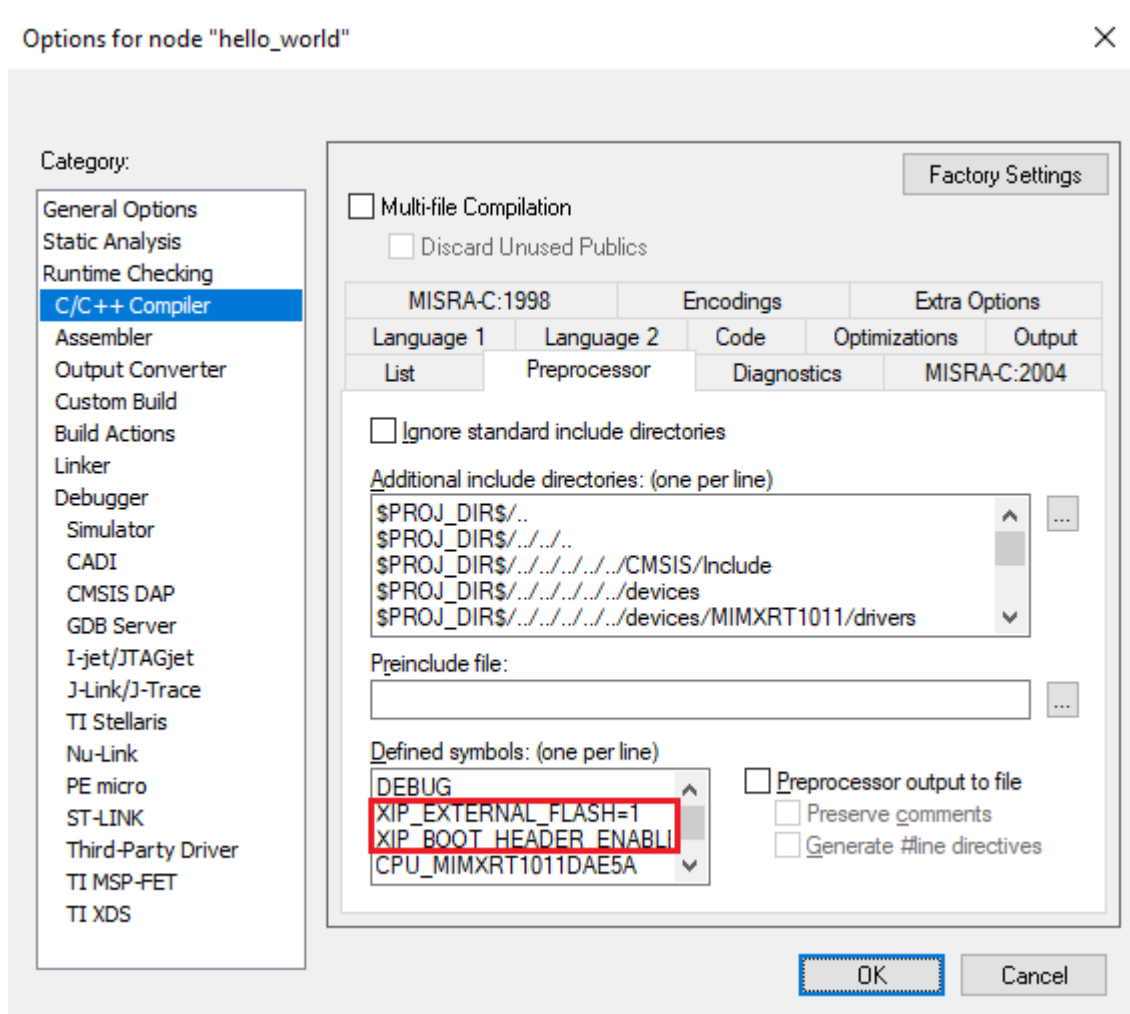
^

| **XIP_EXTERNAL_FLASH=1** | **XIP_BOOT_HEADER_ENABLE=1** | Can be programmed to external flash by IDE and can run after POR reset if external flash is the boot source. | | **XIP_BOOT_HEADER_ENABLE=0** | Can NOT run after POR reset if it is programmed by IDE even if external flash is the boot source. | | **XIP_EXTERNAL_FLASH=0** | This image can NOT do XIP because when this macro is set to 1, it will exclude the code which will change the clock of flexspi. |

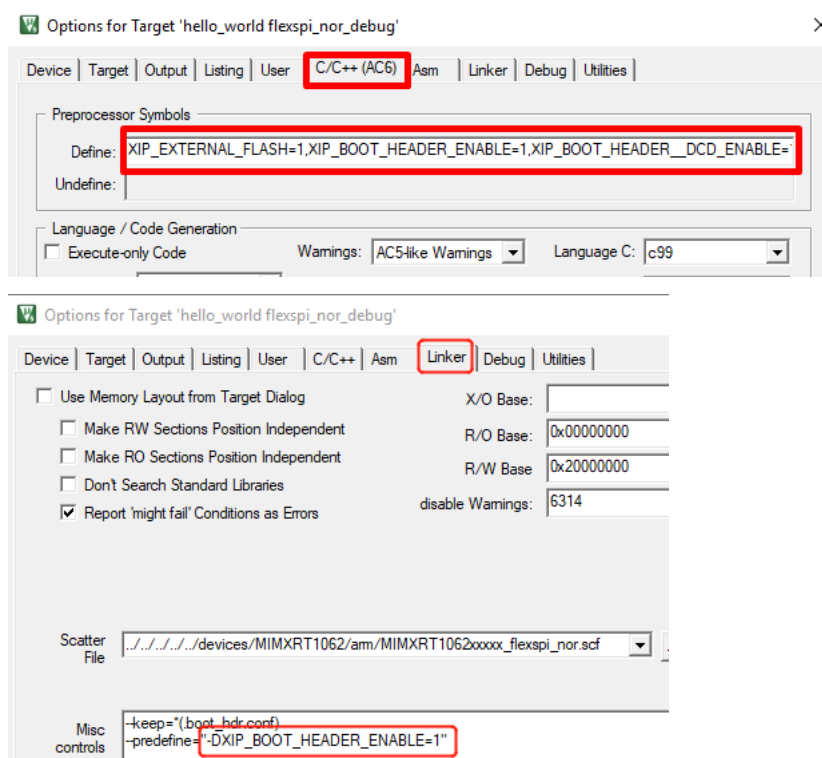
Where to change the macros for each toolchain in MCUXpresso SDK?

Take `hello_world` as an example:

- IAR



• MDK



- **ARMGCC**

Change the configuration in CMakeLists.txt.

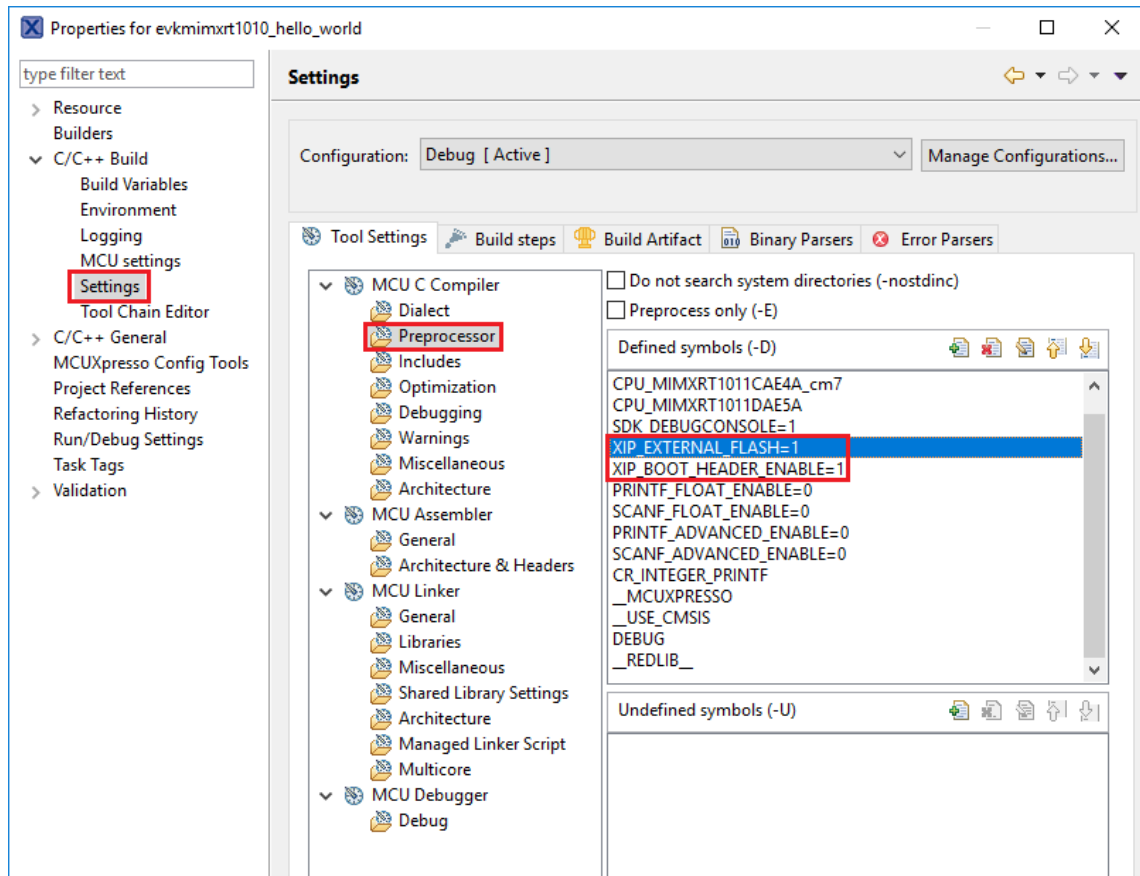
```
SET(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE} -MP")

SET(CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG "${CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG} -DXIP_EXTERNAL_FLASH=1")

SET(CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG "${CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG} -DXIP_BOOT_HEADER_ENABLE=1")

SET(CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG "${CMAKE_C_FLAGS_FLEXSPI_NOR_DEBUG} -DDEBUG")
```

- **MCUX**



1.3 Getting Started with MCUXpresso SDK GitHub

1.3.1 Getting Started with MCUXpresso SDK Repository

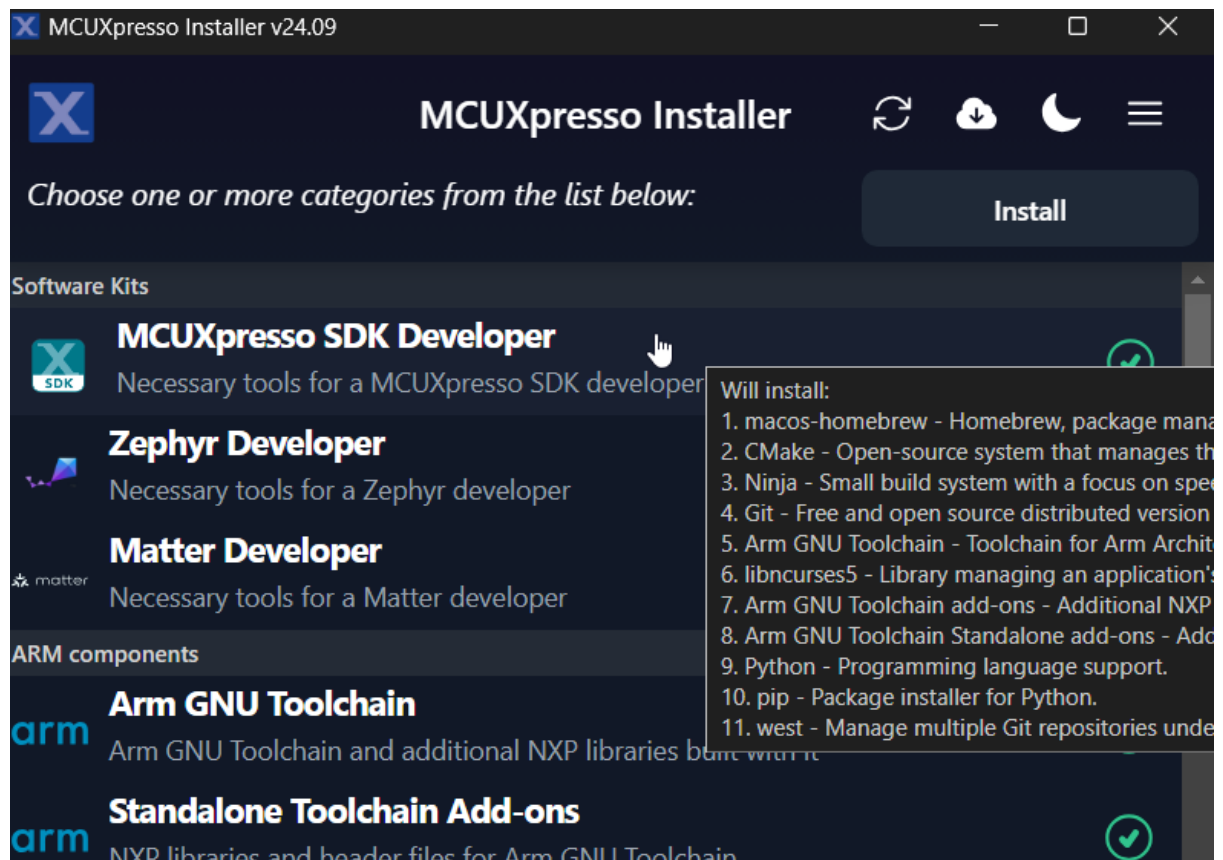
Installation

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_SE	c:\NXP\zephyr-sdk-<version> for windows/opt/zephyr-sdk-<version> for Linux	– toolchain zephyr
CodeWarrior	CW_DIR	C:\Freescale\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:

- arm
- common
- install-info

- 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.

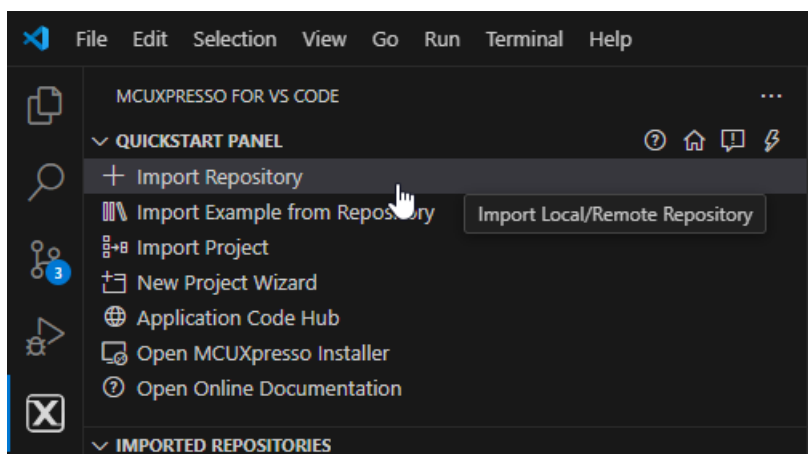
Run a demo using MCUXpresso for VS Code

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](#).

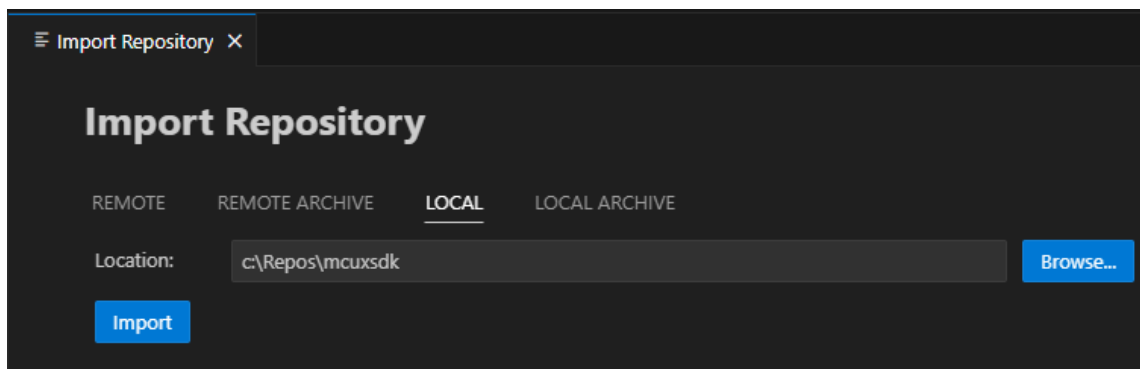
To build an example application:

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

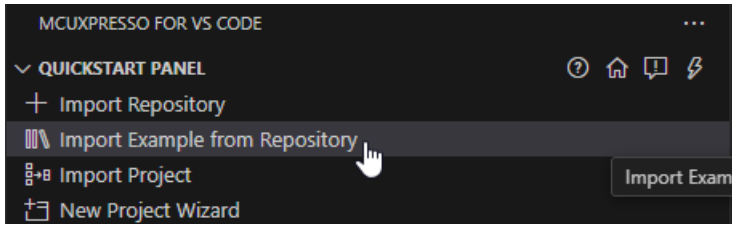


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

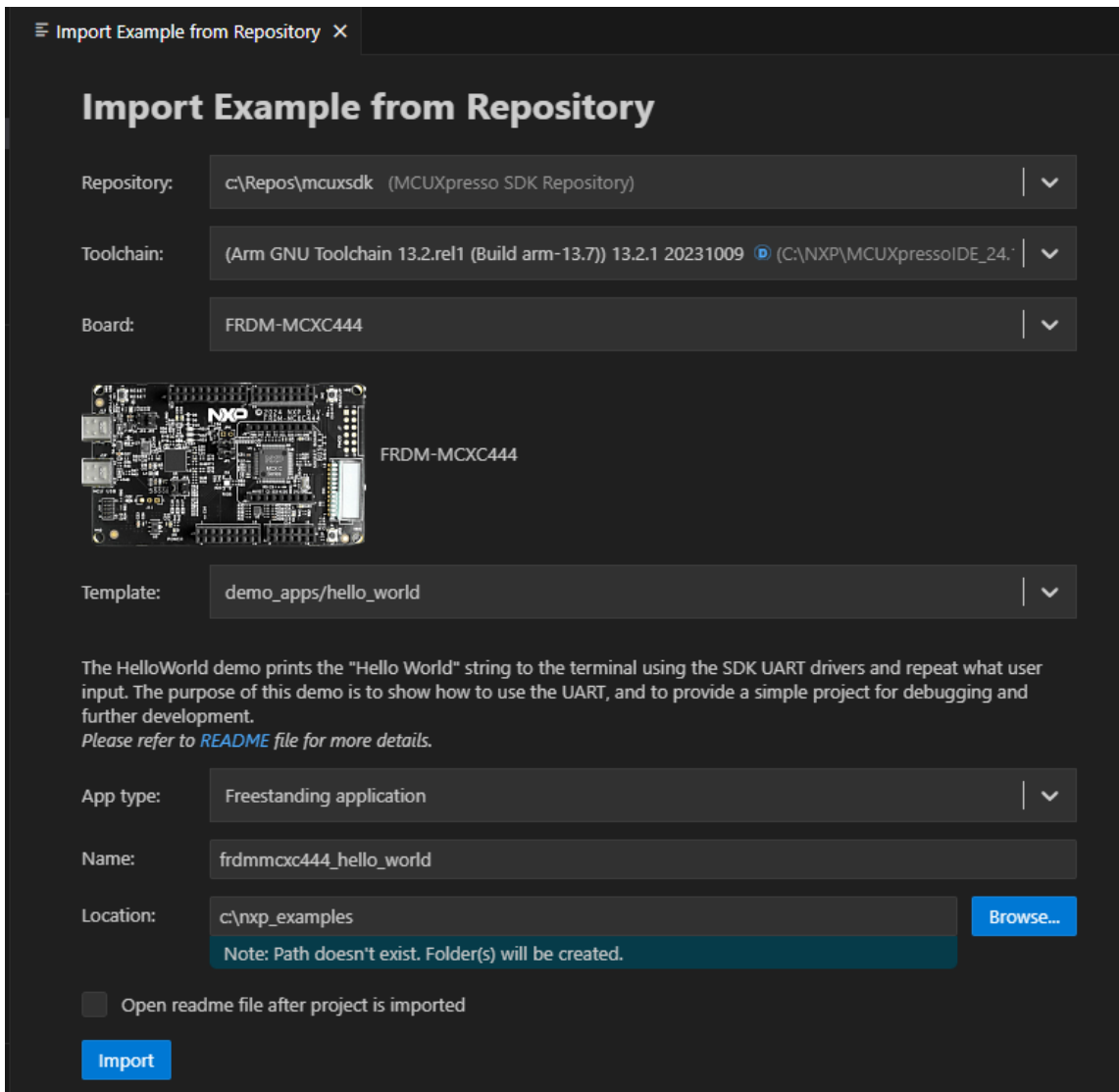
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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

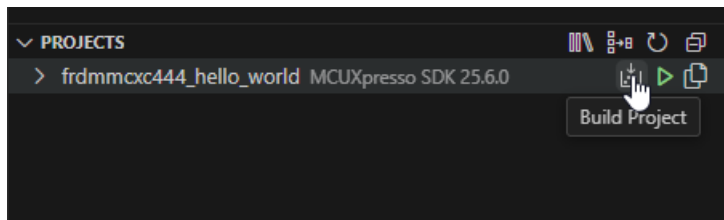


In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.

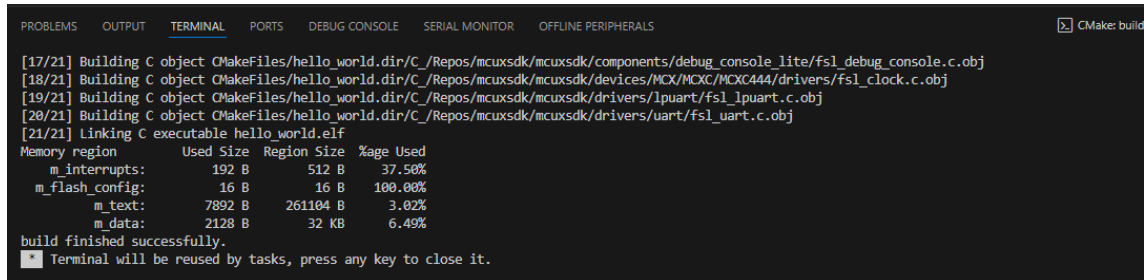


Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Free-standing applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Free-standing examples can be imported 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.

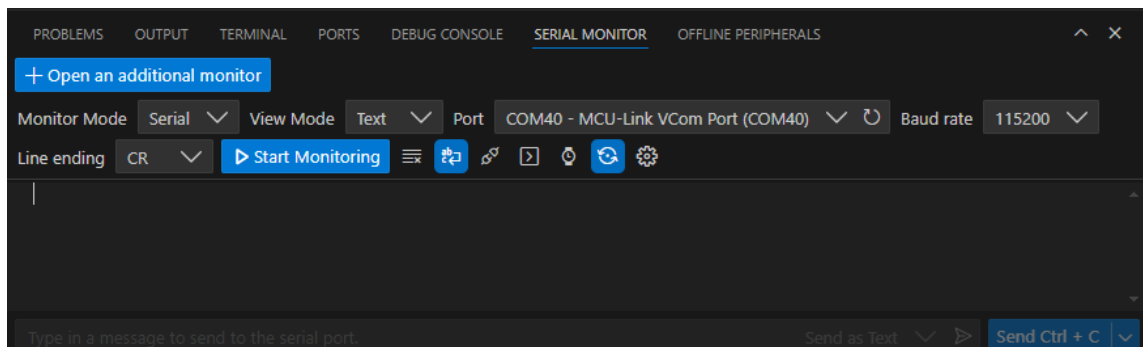


The integrated terminal will open at the bottom and will display the build output.

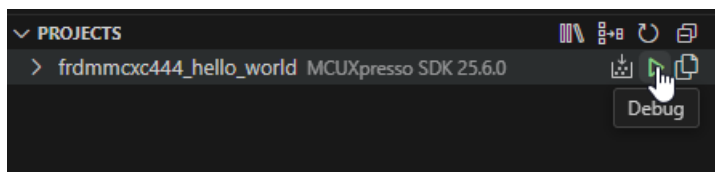


Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

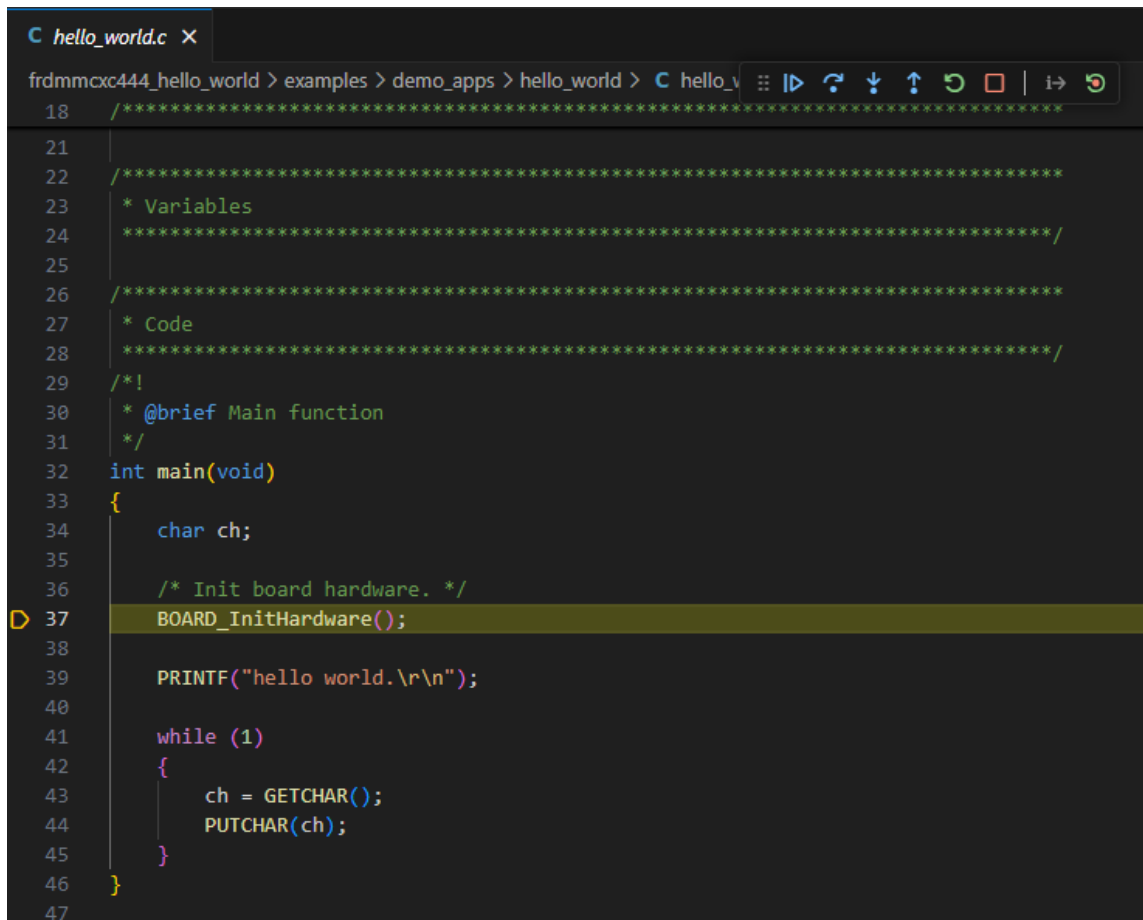
1. Open the **Serial Monitor** from the VS Code's integrated terminal. Select the VCom Port for your device and set the baud rate to 115200.



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.

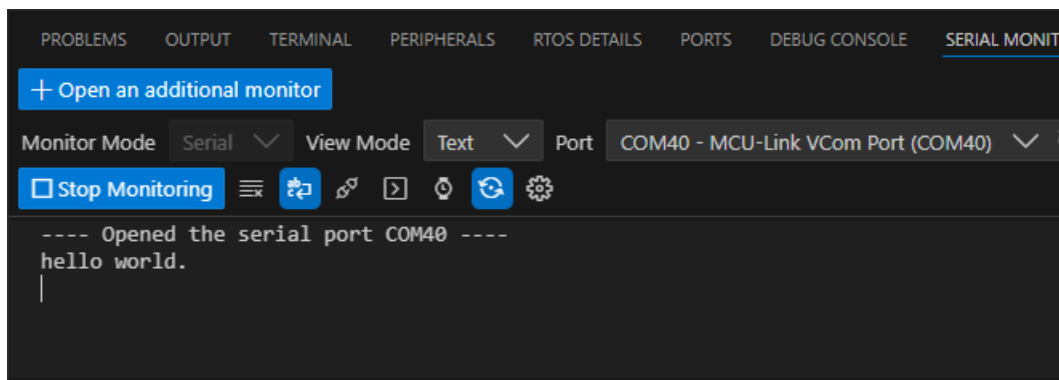


```

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

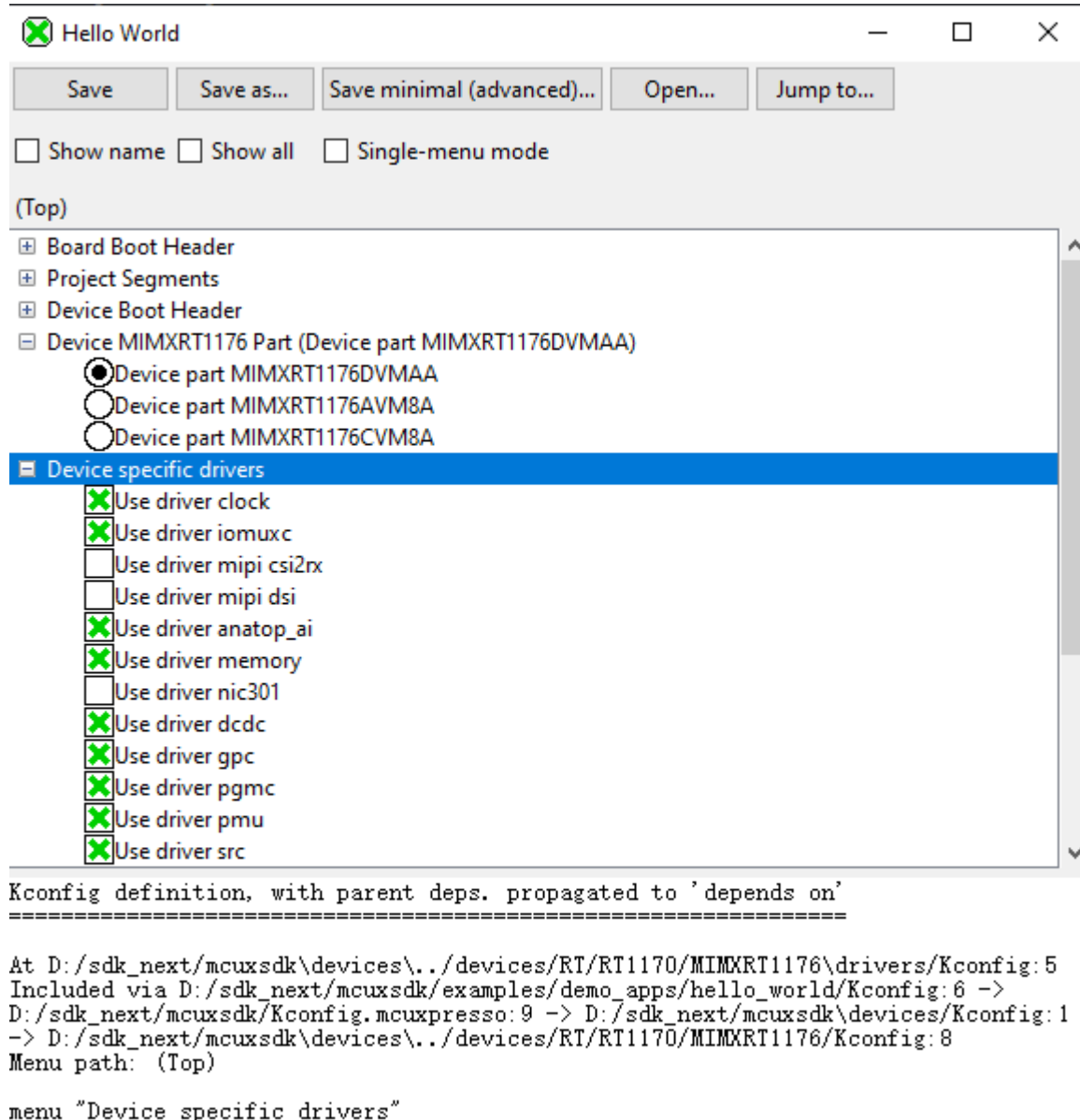
```
west build -b evkbnimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

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:

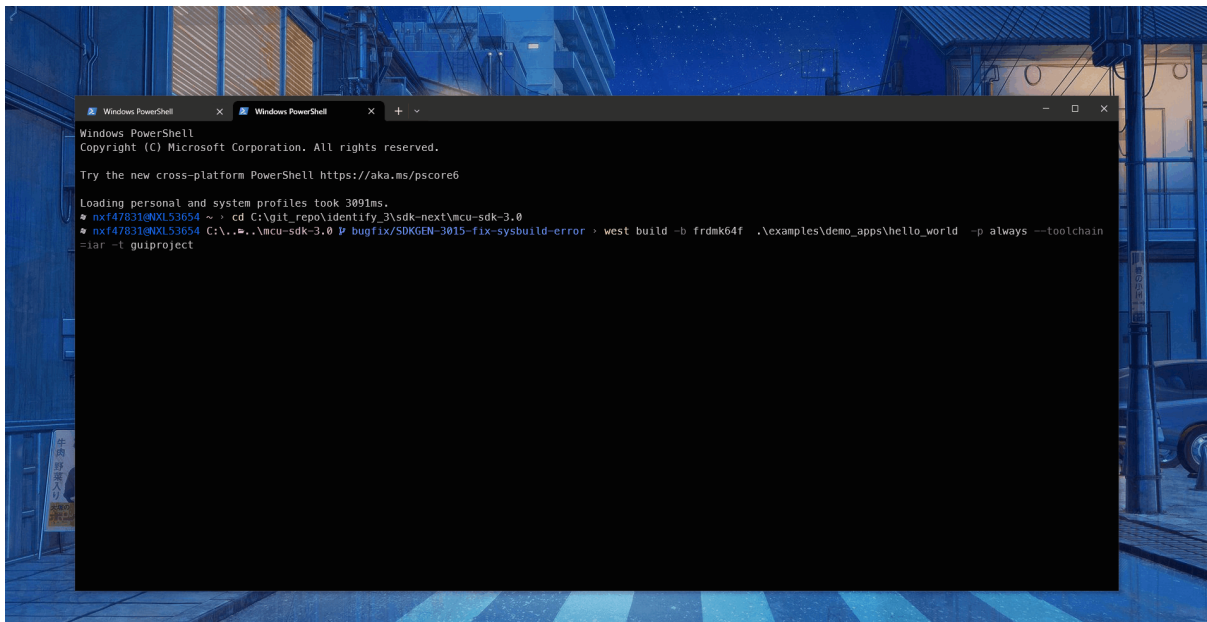
```
west debug -r linkserver
```

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.

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

The MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with Arm Cortex-M-based devices from NXP, including its general purpose, crossover and Bluetooth-enabled MCUs. MCUXpresso SW and Tools for DSC

further extends the SDK support to current 32-bit Digital Signal Controllers. The MCUXpresso SDK includes production-grade software with integrated RTOS (optional), integrated enabling software technologies (stacks and middleware), reference software, and more.

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

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

MCUXpresso SDK

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

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

Development tools

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

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

Supported development systems

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

Development boards	MCU devices
EVK-MIMXRT1010	MIMXRT1011CAE4A, MIMXRT1011DAE5A

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.

USB Type-C PD Stack See the *MCUXpresso SDK USB Type-C PD Stack User's Guide* (document MCUXSDKUSBDUG) for more information

USB Host, Device, OTG Stack See the *MCUXpresso SDK USB Stack User's Guide* (document MCUXSDKUSBSUG) for more information.

TinyCBOR Concise Binary Object Representation (CBOR) Library

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

mbedTLS mbedtls SSL/TLS library v2.x

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

JPEG library JPEG library

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

File systemFatfs The FatFs file system is integrated with the MCUXpresso SDK and can be used to access either the SD card or the USB memory stick when the SD card driver or the USB Mass Storage Device class implementation is used.

emWin The MCUXpresso SDK is pre-integrated with the SEGGER emWin GUI middleware. The AppWizard provides developers and designers with a flexible tool to create stunning user interface applications, without writing any code.

Release contents

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

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

Known issues

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

New Project Wizard compile failure

The following components request the user to manually select other components that they depend upon in order to compile.

These components depend on several other components and the New Project Wizard (NPW) is not able to decide which one is needed by the user.

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

****Components:****issdk_mag3110, issdk_host, systick, gpio_kinetic, gpio_lpc, issdk_mpl3115, sensor_fusion_agm01, sensor_fusion_agm01_lpc, issdk_mma845x, issdk_mma8491q, issdk_mma865x, issdk_mma9553, and CMSIS_RTOS2.CMSIS_RTOS2, and components which include cache driver, such as enet_qos.

Also for low-level adapter components, currently the different types of the same adapter cannot be selected at the same time.

For example, if there are two types of timer adapters, gpt_adapter and pit_adapter, only one can be selected as timer adapter

in one project at a time. Duplicate implementation of the function results in an error.

Note: Most of middleware components have complex dependencies and are not fully supported in new project wizard. Adding a middleware component may result in compile failure.

CMSIS PACK new project compile failure

The generated configuration cannot be applied globally. The components, serial_manager_usb_cdc_virtual and serial_manager_usb_cdc_virtual_xxx (xxx means core variants like cm0plus, cm33, cm4, and cm33_nodsp) are unsupported for new project wizard of CMSIS pack and will lead to compile failure if selected while creating new project(s).

Non XIP target debug issue on toolchain MDK

When debugging non XIP targets in flash boot mode, if application changes any settings which have impacts on flexspi, the build output window might show “Debug access failed” when start debugging next time. It is recommended to keep the board in serial downloader mode when debugging non XIP targets.

Safety_iec60730b cloned project fails to build

When you use the MCUXpresso Config Tool to clone the “safety_iec60730b” project in MCUXpresso SDK package, the created project fails to build.

The build fails because the post-build setup for CRC is incorrect. Therefore, It is recommended to use the “safety_iec60730b” project in MCUXpresso SDK package.

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config**[25.06.00]**

- Initial version

pin_mux**[25.06.00]**

- Initial version
-

ADC**[2.0.4]**

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

[2.0.3]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.1 10.4 10.7 17.7.

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

ADC_ETC

[2.3.2]

- Improvements
 - Corrected that FSL_FEATURE_ADC_ETC_HAS_NO_TSC1_TRIG should be used instead of FSL_FEATURE_ADC_ETC_HAS_NO_TSC0_TRIG in some places.
 - For ADC_ETC without TSC trigger source, CTRL [bit 30] shall be cleared explicitly.

[2.3.1]

- Improvements
 - Change ADC_ETC default DMA Mode to kADC_ETC_TrigDMAWithPulsedSignal. Generally speaking, DMA transfer requests should only be cleared by DMA ACK, and the CPU should not clear the request source. If some users use option kADC_ETC_TrigDMAWithLatchedSignal, changing the mode to kADC_ETC_TrigDMAWithPulsedSignal also meet their requirements.

[2.3.0]

- Improvements
 - Added blocking way to implement SW trigger.

[2.2.1]

- Improvements
 - Modified macro “ADC_ETC_DONE2_ERR_IRQ_TRIG0_DONE2_MASK” to “ADC_ETC_DONE2_3_ERR_IRQ_TRIG0_DONE2_MASK” based on the updates of header file.

[2.2.0]

- Improvements
 - Defined two macros to support some devices that do not equipped with TSC trigger.

[2.1.1]

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

[2.1.0]

- New Features
 - Supported independent IRQ enable bit in ADC-ETC chain configuration registers.
 - Supported trigger n DONE3 interrupt operations.
- Bug Fixes
 - Fixed the violation of MISRA-2012 rules:
 - * Rule 10.1 10.3 10.7 15.5 16.1 16.3 16.4 17.7

[2.0.1]

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

[2.0.0]

- Initial version.
-

AIPSTZ

[2.0.1]

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

[2.0.0]

- Initial version.
-

AOI

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.8, 2.2.

[2.0.0]

- Initial version.
-

CACHE ARMv7-M7

[2.0.4]

- Bug Fixes
 - Fixed doxygen issue.

[2.0.3]

- Improvements
 - Deleted redundancy code about calculating cache clean/invalidate size and address aligns.

[2.0.2]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 10.1, 10.3 and 10.4.

[2.0.1]

- Bug Fixes
 - Fixed cache size issue in L2CACHE_GetDefaultConfig API.

[2.0.0]

- Initial version.
-

CLOCK

[2.5.3]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rule 10.7.

[2.5.2]

- Bug Fixes
 - Fixed issues in CLOCK_GetSysPfdFreq() and CLOCK_GetUsb1PfdFreq() which produce incorrect result.

[2.5.1]

- Improvements
 - Added enumeration clock_div_value_t.

[2.5.0]

- New Features
 - Added CLOCK_IsUsb1PfdEnabled and CLOCK_IsSysPfdEnabled to get the clock source status.
- Bug Fixes
 - Fixed the wrong mxu enumerator used in “CLOCK_ROOT_MUX_TUPLE” macro.
 - Updated CLOCK_SetClockOutput1() function, adjusted the sequence of register configuration.

[2.4.1]

- Improvements
 - Placed function internal constants into initialized data segment.

[2.4.0]

- New Features
 - Added clock output related APIs and data structures.
 - Added one function `CLOCK_GetClockRootFreq` to get the frequency of each clock root.

[2.3.1]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rule10.1, rule 18.4, rule 14.4 and so on.

[2.3.0]

- New feature:
 - Moved `SDK_DelayAtLeastUs` function from clock driver to common driver.

[2.2.0]

- New feature
- Adding new API `CLOCK_DelayAtLeastUs()` implemented by DWT to allow users set delay in unit of microsecond.

[2.1.6]

- Bug Fix:
 - Fix build issue with GCC compiler when include header from C++ file.

[2.1.5]

- Bug Fix:
 - Add initialization of the fractional mode and spread spectrum mode in `CLOCK_InitSysPll()`.

[2.1.4]

- Optimization:
 - Add `PerClk` in `clock_name_t` and `CLOCK_GetFreq`.
 - Add APIs to get the frequency of AHB clock and SEMC, IPG clock and PER clock.

[2.1.3]

- Use double instead of `uint64_t` to achieve better performance with double precision FPU.

[2.1.2]

- some minor fixes.

[2.0.0]

- initial version.
-

COMMON

[2.6.1]

- Improvements
 - Support Cortex M23.

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.3]

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

[2.3.2]

- Improvements
 - Make driver aarch64 compatible

[2.3.1]

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

[2.3.0]

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

[2.2.10]

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

[2.2.9]

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

[2.2.8]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

DCDC

[2.3.0]

- Improvements
 - REG3[MISC_DELAY_TIMING], REG2[LOOPCTRL_DC_R], and REG2[LOOPCTRL_DC_C] are reserved in the latest RM, deleted corresponding functions.

[2.2.1]

- Improvements
 - Fixed the doxygen warning.

[2.2.0]

- New Features
 - Added supports for i.MXRT1170 series.
- Bug Fixes
 - Fixed the warning that the `DCDC_ConvertByteArrayToWorld` function defined but not used.
- Improvements
 - Updated `rcscale` to reduce the ripple when booting into DCM.

[2.1.0]

- Improvements
 - Divided the `DCDC_AdjustTargetVoltage()` into two APIs for two different modes.
- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.1, 10.4, 16.4, 17.7.

[2.0.0]

- Initial version.
-

DCP

[2.1.7]

- Bug Fix
 - Reduce optimization level for critical functions working with SRF.

[2.1.6]

- Bug Fix
 - MISRA C-2012 issue fix.

[2.1.5]

- Improvements
 - Added support when DCACHE enabled. Input and output buffers should be in non-cached memory or handled properly (DCACHE Clean and Invalidate).

[2.1.4]

- Bug Fix
 - Fix CRC-32 computation issue on the code's block boundary size.

[2.1.3]

- Bug Fix
 - MISRA C-2012 issue fixed: rule 10.1, 10.3, 10.4, 11.9, 14.4, 16.4 and 17.7.

[2.1.2]

- Bug Fix
 - Fix sign-compare warning in dcp_reverse_and_copy.

[2.1.1]

- Improvements
 - Added DCP status clearing when channel operation is complete.

[2.1.0]

- New Features
 - Added byte/word swap feature for key, input, and output data.

[2.0.0]

- Initial version.
-

DMAMUX**[2.1.3]**

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - Modify the type of parameter source from uint32_t to int32_t in the DMA-MUX_SetSource.

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

EDMA

[2.4.7]

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

[2.4.6]

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

[2.4.5]

- Bug Fixes
 - Fixed memory convert would convert NULL as zero address issue.

[2.4.4]

- Bug Fixes
 - Fixed comments by replacing STCD with TCD
 - Fixed the TCD overwrite issue when submit transfer request in the callback if there is a active TCD in hardware.
 - Fixed violations of MISRA C-2012 rule 10.8,5.6.

[2.4.3]

- Improvements
 - Added FSL_FEATURE_MEMORY_HAS_ADDRESS_OFFSET to convert the address between system mapped address and dma quick access address.
- Bug Fixes
 - Fixed the wrong tcd done count calculated in first TCD interrupt for the non scatter gather case.

[2.4.2]

- Bug Fixes
 - Fixed the wrong tcd done count calculated in first TCD interrupt by correct the initial value of the header.
 - Fixed violations of MISRA C-2012 rule 10.3, 10.4.

[2.4.1]

- Bug Fixes
 - Added clear CITER and BITER registers in EDMA_AbortTransfer to make sure the TCD registers in a correct state for next calling of EDMA_SubmitTransfer.
 - Removed the clear DONE status for ESG not enabled case to avoid DONE bit cleared unexpectedly.

[2.4.0]

- Improvements
 - Added api EDMA_EnableContinuousChannelLinkMode to support continuous link mode.
 - Added apis EDMA_SetMajorOffsetConfig/EDMA_TcdSetMajorOffsetConfig to support major loop address offset feature.
 - Added api EDMA_EnableChannelMinorLoopMapping for minor loop offset feature.
 - Removed the redundant IRQ Handler in edma driver.

[2.3.2]

- Improvements
 - Fixed HIS ccm issue in function EDMA_PrepareTransferConfig.
 - Fixed violations of MISRA C-2012 rule 11.6, 10.7, 10.3, 18.1.
- Bug Fixes

- Added ACTIVE & BITER & CITER bitfields to determine the channel status to fixed the issue of the transfer request cannot submit by function EDMA_SubmitTransfer when channel is idle.

[2.3.1]

- Improvements
 - Added source/destination address alignment check.
 - Added driver IRQ handler support for multi DMA instance in one SOC.

[2.3.0]

- Improvements
 - Added new api EDMA_PrepareTransferConfig to allow different configurations of width and offset.
- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.4, 10.1.
 - Fixed the Coverity issue regarding out-of-bounds write.

[2.2.0]

- Improvements
 - Added peripheral-to-peripheral support in EDMA driver.

[2.1.9]

- Bug Fixes
 - Fixed MISRA issue: Rule 10.7 and 10.8 in function EDMA_DisableChannelInterrupts and EDMA_SubmitTransfer.
 - Fixed MISRA issue: Rule 10.7 in function EDMA_EnableAsyncRequest.

[2.1.8]

- Bug Fixes
 - Fixed incorrect channel preemption base address used in EDMA_SetChannelPreemptionConfig API which causes incorrect configuration of the channel preemption register.

[2.1.7]

- Bug Fixes
 - Fixed incorrect transfer size setting.
 - * Added 8 bytes transfer configuration and feature for RT series;
 - * Added feature to support 16 bytes transfer for Kinetis.
 - Fixed the issue that EDMA_HandleIRQ would go to incorrect branch when TCD was not used and callback function not registered.

[2.1.6]

- Bug Fixes
 - Fixed KW3X MISRA Issue.
 - * Rule 14.4, 10.8, 10.4, 10.7, 10.1, 10.3, 13.5, and 13.2.
- Improvements
 - Cleared the IRQ handler unavailable for specific platform with macro `FSL_FEATURE_EDMA_MODULE_CHANNEL_IRQ_ENTRY_SHARED_OFFSET`.

[2.1.5]

- Improvements
 - Improved EDMA IRQ handler to support half interrupt feature.

[2.1.4]

- Bug Fixes
 - Cleared enabled request, status during `EDMA_Init` for the case that EDMA is halted before reinitialization.

[2.1.3]

- Bug Fixes
 - Added clear DONE bit in IRQ handler to avoid overwrite TCD issue.
 - Optimized above solution for the case that transfer request occurs in callback.

[2.1.2]

- Improvements
 - Added interface to get next TCD address.
 - Added interface to get the unused TCD number.

[2.1.1]

- Improvements
 - Added documentation for eDMA data flow when scatter/gather is implemented for the `EDMA_HandleIRQ` API.
 - Updated and corrected some related comments in the `EDMA_HandleIRQ` API and `edma_handle_t` struct.

[2.1.0]

- Improvements
 - Changed the `EDMA_GetRemainingBytes` API into `EDMA_GetRemainingMajorLoopCount` due to eDMA IP limitation (see API comments/note for further details).

[2.0.5]

- Improvements
 - Added pubweak DriverIRQHandler for K32H844P (16 channels shared).

[2.0.4]

- Improvements
 - Added support for SoCs with multiple eDMA instances.
 - Added pubweak DriverIRQHandler for KL28T DMA1 and MCIMX7U5_M4.

[2.0.3]

- Bug Fixes
 - Fixed the incorrect pubweak IRQHandler name issue, which caused re-definition build errors when client set his/her own IRQHandler, by changing the 32-channel IRQHandler name to DriverIRQHandler.

[2.0.2]

- Bug Fixes
 - Fixed incorrect minorLoopBytes type definition in `_edma_transfer_config` struct, and defined `minorLoopBytes` as `uint32_t` instead of `uint16_t`.

[2.0.1]

- Bug Fixes
 - Fixed the eDMA callback issue (which did not check valid status) in `EDMA_HandleIRQ` API.

[2.0.0]

- Initial version.
-

EWM

[2.0.4]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed the hard fault in EWM_Deinit.

[2.0.0]

- Initial version.
-

FLEXIO**[2.3.0]**

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Improvements
 - Added doxygen index parameter comment in FLEXIO_SetClockMode.

[2.2.0]

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

[2.1.0]

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

[2.0.4]

- Bug Fixes
 - Fixed MISRA 8.4 issues.

[2.0.3]

- Bug Fixes
 - Fixed MISRA 10.4 issues.

[2.0.2]

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

[2.0.1]

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

FLEXIO_I2C

[2.6.2]

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues

[2.6.0]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

- Bug Fixes
 - Fixed the FLEXIO I2C issue where the master could not receive data from I2C slave in high baudrate.
 - Fixed the FLEXIO I2C issue where the master could not receive NAK when master sent non-existent addr.
 - Fixed the FLEXIO I2C issue where the master could not get transfer count successfully.
 - Fixed the FLEXIO I2C issue where the master could not receive data successfully when sending data first.
 - Fixed the Dozen mode configuration error in FLEXIO_I2C_MasterInit API. For enableInDoze = true, the configuration should be 0; for enableInDoze = false, the configuration should be 1.

- Fixed the issue that FLEXIO_I2C_MasterTransferBlocking API called FLEXIO_I2C_MasterTransferCreateHandle, which lead to the s_flexioHandle/s_flexioIsr/s_flexioType variable being written. Then, if calling FLEXIO_I2C_MasterTransferBlocking API multiple times, the s_flexioHandle/s_flexioIsr/s_flexioType variable would not be written any more due to it being out of range. This lead to the following situation: NonBlocking transfer APIs could not work due to the fail of register IRQ.

[2.1.1]

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

[2.1.0]

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

FLEXIO_I2S

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_I2S_EDMA

[2.1.9]

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

[2.1.8]

- Improvements
 - Applied EDMA ERRATA 51327.
-

FLEXIO_SPI**[2.4.3]**

- Improvements
 - Make SPI_RETRY_TIMES configurable by CONFIG_SPI_RETRY_TIMES.

[2.4.2]

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

[2.4.1]

- Bug Fixes
 - Fixed coverity issues

[2.4.0]

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

[2.3.5]

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

[2.3.4]

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

[2.3.3]

- Bugfixes
 - Fixed cs-continuous mode.

[2.3.2]

- Improvements
 - Changed FLEXIO_SPI_DUMMYDATA to 0x00.

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.3]

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

[2.1.2]

- Bug Fixes
 - The following modification support FlexIO using multiple instances:
 - * Removed FLEXIO_Reset API in module Init APIs.

- * Updated module Deinit APIs to reset the shifter/timer config instead of disabling module/clock.
- * Updated module Enable APIs to only support enable operation.

[2.1.1]

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

[2.1.0]

- New Features
 - Added Transfer prefix in transactional APIs.
 - Added transferSize in handle structure to record the transfer size.
 - Bug Fixes
 - Fixed the error register address return for 16-bit data write in FLEXIO_SPI_GetTxDataRegisterAddress.
 - Provided independent IRQHandler/transfer APIs for Master and slave to fix the baudrate limit issue.
-

FLEXIO_UART

[2.6.4]

- Improvements
 - Make UART_RETRY_TIMES configurable by CONFIG_UART_RETRY_TIMES.

[2.6.3]

- Bug Fixes
 - Fixed coverity issues

[2.6.2]

- Bug Fixes
 - Fixed coverity issues

[2.6.1]

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

[2.6.0]

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

[2.5.1]

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

[2.5.0]

- Improvements
 - Added API FLEXIO_UART_FlushShifters to flush UART fifo.

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_UART_EDMA

[2.3.1]

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

[2.3.0]

- Refer FLEXIO_UART driver change log to 2.3.0
-

FLEXRAM

[2.3.0]

- New Features
 - Supported platforms which have ECC but no ECC error injection.

[2.2.0]

- New Features
 - Supported flexram ECC error injection function.

[2.1.0]

- New Features
 - Supported flexram ECC function.

[2.0.7]

- Bug Fixes
 - Fixed doxygen issue.

[2.0.6]

- New Features
 - Updated bank configuration and TCM size with GPR16/GPR17/GPR18 into SOC level for different SOC.

[2.0.5]

- New Features
 - Added the magic address feature for OCRAM, DTCM and ITCM.

[2.0.4]

- Bug Fixes
 - Fixed FlexRAM driver's missing extern C around functions in header file.
 - Removed magic address feature from driver.

[2.0.3]

- Bug Fixes
 - Fixed the issue that TCM size configuration was wrong when TCM bank number was not a value power of 2.

[2.0.2]

- Bug Fixes
 - Updated driver due to Reference Manual update.

[2.0.1]

- Bug Fixes
 - Fixed MISRA issue.

[2.0.0]

- Initial version.
-

FLEXSPI

[2.8.1]

- Improvements
 - Updated the LUT configuration parameter checking with flexible way to adapt different Socs.

[2.8.0]

- Bug Fixes
 - Introduced the **disableAhbReadResume** field in the flexspi_config_t structure to provide control over the AHBCR[RESUMEDISABLE] register bit.
 - Implemented a workaround for hardware erratum ERR052733 by setting the default value of **disableAhbReadResume** to **true**.
 - Fixed issue in FLEXSPI_TransferHandleIRQ where the transfer completion was incorrectly signaled despite pending read/write operations.

- New Features
 - Introduced a new function(`FLEXSPI_UpdateAhbBuffersSettings`) that allows users to update the AHB buffer configuration after the FLEXSPI module has been initialized

[2.7.0]

- New Features
 - Added new API to support address remapping.

[2.6.4]

- Improvements
 - Added new macro “`FSL_SDK_ENABLE_FLEXSPI_RESET_CONTROL`” to support driver level reset control.

[2.6.3]

- Bug Fixes
 - Fixed an issue which cause `IPCR1[IPAREN]` cleared by mistake.

[2.6.2]

- Bug Fixes
 - Wait Bus IDLE before operation of `FLEXSPI_SoftwareReset()`, `FLEXSPI_TransferBlocking()` and `FLEXSPI_TransferNonBlocking()`.

[2.6.1]

- Bug Fixes
 - Updated code of reset peripheral.
 - Updated `FLEXSPI_UpdateLUT()` to check if input lut address is not in Flexspi AMBA region.
 - Updated `FLEXSPI_Init()` to check if input AHB buffer size exceeded maximum AHB size.

[2.6.0]

- New Features
 - Added new API to set AHB memory-mapped flash base address.
 - Added support of `DLLxCR[REFPHASEGAP]` bit field, it is recommended to set it as 0x2 if DLL calibration is enabled.

[2.5.1]

- Bugfixes
 - Fixed handling of W1C bits in the INTR register
 - Removed FIFO resets from `FLEXSPI_CheckAndClearError`
 - `FLEXSPI_TransferBlocking` is observing `IPCMDDONE` and then fetches the final status of the transfer

- Fixed issue that FLEXSPI2_DriverIRQHandler not defined.

[2.5.0]

- Improvements
 - Supported word un-aligned access for write/read blocking/non-blocking API functions.
 - Fixed dead loop issue in DLL update function when using FRO clock source.
 - Fixed violations of the MISRA C-2012 Rule 10.3.

[2.4.0]

- Improvements
 - Isolated IP command parallel mode and AHB command parallel mode using feature MACRO.
 - Supported new column address shift feature for external memory.

[2.3.5]

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

[2.3.4]

- Bug Fixes
 - Updated flexspi_config_t structure and FlexSPI_Init to support new feature FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_COMBINATION.

[2.3.3]

- Bug Fixes
 - Removed feature FSL_FEATURE_FLEXSPI_DQS_DELAY_PS for DLL delay setting. Changed to use feature FSL_FEATURE_FLEXSPI_DQS_DELAY_MIN to set slave delay target as 0 for DLL enable and clock frequency higher than 100MHz.

[2.3.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 8.4, 8.5, 10.1, 10.3, 10.4, 11.6 and 14.4.

[2.3.1]

- Bug Fixes
 - Wait for bus to be idle before using it as access to external flash with new setting in FLEXSPI_SetFlashConfig() API.
 - Fixed the potential buffer overread and Tx FIFO overwrite issue in FLEXSPI_WriteBlocking.

[2.3.0]

- New Features
 - Added new API FLEXSPI_UpdateDllValue for users to update DLL value after updating flexspi root clock.
 - Corrected grammatical issues for comments.
 - Added support for new feature FSL_FEATURE_FLEXSPI_DQS_DELAY_PS in DLL configuration.

[2.2.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3 and 10.4.
 - Updated _flexspi_command from named enumerator into anonymous enumerator.

[2.2.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.8, 11.9, 14.4, 15.7, 16.4, 17.7, 7.3.
 - Fixed IAR build warning Pe167.
 - Fixed the potential buffer overwrite and Rx FIFO overread issue in FLEXSPI_ReadBlocking.

[2.2.0]

- Bug Fixes
 - Fixed flag name typos: kFLEXSPI_IpTxFifoWatermarkEmpltyFlag to kFLEXSPI_IpTxFifoWatermarkEmptyFlag; kFLEXSPI_IpCommandExcutionDoneFlag to kFLEXSPI_IpCommandExecutionDoneFlag.
 - Fixed comments typos such as sequencen->sequence, levle->level.
 - Fixed FLSHCR2[ARDSEQID] field clean issue.
 - Updated flexspi_config_t structure and FlexSPI_Init to support new feature FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_ATDFEN and FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_ARDFEN.
 - Updated flexspi_flags_t structure to support new feature FSL_FEATURE_FLEXSPI_HAS_INTEN_AHBBUSERROREN.

[2.1.1]

- Improvements
 - Defaulted enable prefetch for AHB RX buffer configuration in FLEXSPI_GetDefaultConfig, which is align with the reset value in AHB RX BUFxCR0.
 - Added software workaround for ERR011377 in FLEXSPI_SetFlashConfig; added some delay after DLL lock status set to ensure correct data read/write.

[2.1.0]

- New Features
 - Added new API FLEXSPI_UpdateRxSampleClock for users to update read sample clock source after initialization.
 - Added reset peripheral operation in FLEXSPI_Init if required.

[2.0.5]

- Bug Fixes
 - Fixed FLEXSPI_UpdateLUT cannot do partial update issue.

[2.0.4]

- Bug Fixes
 - Reset flash size to zero for all ports in FLEXSPI_Init; fixed the possible out-of-range flash access with no error reported.

[2.0.3]

- Bug Fixes
 - Fixed AHB receive buffer size configuration issue. The FLEXSPI_AHBRXBUFFCRO_BUFSZ field should configure 64 bits size, and currently the AHB receive buffer size is in bytes which means 8-bit, so the correct configuration should be `config->ahbConfig.buffer[i].bufferSize / 8`.

[2.0.2]

- New Features
 - Supported DQS write mask enable/disable feature during set FLEXSPI configuration.
 - Provided new API FLEXSPI_TransferUpdateSizeEDMA for users to update eDMA transfer size(SSIZE/DSIZE) per DMA transfer.
- Bug Fixes
 - Fixed invalid operation of FLEXSPI_Init to enable AHB bus Read Access to IP RX FIFO.
 - Fixed incorrect operation of FLEXSPI_Init to configure IP TX FIFO watermark.

[2.0.1]

- Bug Fixes
 - Fixed the flag clear issue and AHB read Command index configuration issue in FLEXSPI_SetFlashConfig.
 - Updated FLEXSPI_UpdateLUT function to update LUT table from any index instead of previous command index.
 - Added bus idle wait in FLEXSPI_SetFlashConfig and FLEXSPI_UpdateLUT to ensure bus is idle before any change to FlexSPI controller.
 - Updated interrupt API FLEXSPI_TransferNonBlocking and interrupt handle flow FLEXSPI_TransferHandleIRQ.
 - Updated eDMA API FLEXSPI_TransferEDMA.

[2.0.0]

- Initial version.
-

FLEXSPI EDMA Driver

[2.3.3]

- Bug Fixes
 - Fixed FLEXSPI_TransferEDMA bug that, the DMA channel not configured correctly when using kFLEXSPI_Read.

[2.3.2]

- Bug Fixes
 - Fixed the bug that internal variable s_edmaPrivateHandle overflows when using FlexSPI2.

[2.0.2]

- New Features
 - Provided new API FLEXSPI_TransferUpdateSizeEDMA for users to update eDMA transfer size(SSIZE/DSIZE) per DMA transfer.

[2.0.0]

- Initial version.
-

GPC

[2.1.1]

- Bug Fixes
 - Moved the assert sentence that irq register number has to be greater than 0 to platforms which irq 0-31 is not available.
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.7 12.2.

[2.1.0]

- Improvements
 - Updated driver for IMXRT.

[2.0.0]

- Initial version.
-

GPIO

[2.0.7]

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

[2.0.6]

- Bug Fixes
 - Fixed compile warning: ‘GPIO_GetInstance’ defined but not used when macro FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL is defined.

[2.0.5]

- Bug Fixes
 - Fixed MISRA C-2012 issue: rule-17.7.

[2.0.4]

- Improvements
 - Updated the GPIO_PinWrite to use atomic operation if possible.
- Bug Fixes
 - Fixed GPIO_PortToggle bug with platforms don’t have register DR_TOGGLE.

[2.0.3]

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

[2.0.2]

- Bug Fixes
 - Fixed the bug of enabling wrong GPIO clock gate in initial API. Since some GPIO instances may not have a clock gate enabled, it checks the clock gate number and makes sure the clock gate is valid.

[2.0.1]

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

[2.0.0]

- Initial version.
-

GPT

[2.0.6]

- Bug Fixes
 - Fix CERT INT30-C issues.

[2.0.5]

- Improvements
 - Support workaround for ERR003777. This workaround helps switching the clock sources.

[2.0.4]

- Bug Fixes
 - Fixed compiler warning when built with FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL flag enabled.

[2.0.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 5.3 by customizing function parameter.

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.8, 17.7.

[2.0.0]

- Initial version.
-

IOMUXC

[2.0.2]

- Doxygen improvement.

[2.0.1]

- Delete enum value kIOMUXC_GPR_USBExposureMode in the _iomuxc_gpr_mode.

[2.0.0]

- initial version.
-

KPP**[2.1.1]**

- Bug Fixes
 - Fixed coverity MSG issues with CERT INT30-C, CERT ARR30-C compliance.

[2.1.0]

- Improvements
 - Optimize rowData debounce method to adapt to multi-key detection
 - Modify the KPP_keyPressScanning type to status_t.

[2.0.1]

- Bug Fixes
 - Fixed the violations of MISRA 2012 rules:
 - * Rule 10.3 10.4 10.6 14.4 17.7

[2.0.0]

- Initial version.
-

LPI2C**[2.6.2]**

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues.

[2.6.0]

- New Feature
 - Added common IRQ handler entry LPI2C_DriverIRQHandler.

[2.5.7]

- Improvements
 - Added support for separated IRQ handlers.

[2.5.6]

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

[2.5.5]

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

[2.5.4]

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

[2.5.3]

- Improvements
 - Added handler for LPI2C7 and LPI2C8.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.1]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

- New Features
 - Supported reading more than 256 bytes of data in one transfer as master.
 - Added API LPI2C_GetInstance.
- Bug Fixes
 - Fixed bug in LPI2C_MasterTransferAbortEDMA, LPI2C_MasterTransferAbort and LPI2C_MasterTransferHandleIRQ that before sending stop signal whether master is active and whether stop signal has been sent should be checked, to make sure no FIFO error or bus error will be caused.

- Fixed bug in LPI2C master EDMA transactional layer that the bus error cannot be caught and returned by user callback, by monitoring bus error events in interrupt handler.
- Fixed bug in LPI2C_GetCyclesForWidth that the parameter used to calculate clock cycle should be $2^{\text{prescaler}}$ rather than prescaler.
- Fixed bug in LPI2C_MasterInit that timeout value should be configured after baudrate, since the timeout calculation needs prescaler as parameter which is changed during baudrate configuration.
- Fixed bug in LPI2C_MasterTransferHandleIRQ and LPI2C_RunTransferStateMachine that when master writes with no stop signal, need to first make sure no data remains in the tx FIFO before finishes the transfer.

[2.2.0]

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

[2.1.12]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.1.11]

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

[2.1.10]

- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fixed rule 10.3, 14.4, 15.5.
 - Fixed unaligned access issue in LPI2C_RunTransferStateMachine.
 - Fixed uninitialized variable issue in LPI2C_MasterTransferHandleIRQ.
 - Used linked TCD to disable tx and enable rx in read operation to fix the issue that for platform sharing the same DMA request with tx and rx, during LPI2C read operation if interrupt with higher priority happened exactly after command was sent and before tx disabled, potentially both tx and rx could trigger dma and cause trouble.
 - Fixed MISRA issues.

- * Fixed rules 10.1, 10.3, 10.4, 11.6, 11.9, 14.4, 17.7.
- Fixed the waitTimes variable not re-assignment issue for each byte read.
- New Features
 - Added the IRQHandler for LPI2C5 and LPI2C6 instances.
- Improvements
 - Updated the LPI2C_WAIT_TIMEOUT macro to unified name I2C_RETRY_TIMES.

[2.1.9]

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

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - API name changed:
 - * `LPI2C_MasterTransferCreateHandle` -> `LPI2C_MasterCreateHandle`.

- * LPI2C_MasterTransferGetCount -> LPI2C_MasterGetTransferCount.
- * LPI2C_MasterTransferAbort -> LPI2C_MasterAbortTransfer.
- * LPI2C_MasterTransferHandleIRQ -> LPI2C_MasterHandleInterrupt.
- * LPI2C_SlaveTransferCreateHandle -> LPI2C_SlaveCreateHandle.
- * LPI2C_SlaveTransferGetCount -> LPI2C_SlaveGetTransferCount.
- * LPI2C_SlaveTransferAbort -> LPI2C_SlaveAbortTransfer.
- * LPI2C_SlaveTransferHandleIRQ -> LPI2C_SlaveHandleInterrupt.

[2.0.0]

- Initial version.
-

LPI2C_EDMA

[2.4.5]

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

[2.4.4]

- Improvements
 - Added support for 2KB data transfer

[2.4.3]

- Improvements
 - Added support for separated IRQ handlers.

[2.4.2]

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

[2.4.1]

- Refer LPI2C driver change log 2.0.0 to 2.4.1
-

LPSPi

[2.7.3]

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

[2.7.2]

- Bug Fixes
 - Fixed coverity issues.

[2.7.1]

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

[2.7.0]

- New Feature
 - Added common IRQ handler entry LPSPi_DriverIRQHandler.

[2.6.10]

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

[2.6.9]

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

[2.6.8]

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

[2.6.7]

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

[2.6.6]

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

[2.6.5]

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

[2.6.4]

- Bug Fixes
 - Added LPSPI6_DriverIRQHandler for LPSPI6 instance.

[2.6.3]

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

[2.6.2]

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

[2.6.1]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

- Improvements
 - Improved LPSPI_SlaveTransferHandleIRQ to fill up TX FIFO instead of write one data to TX register which improves the slave transmit performance.
 - Added new functional APIs LPSPI_SelectTransferPCS and LPSPI_SetPCSContinuous to support changing PCS selection and PCS continuous mode.
- Bug Fixes
 - Fixed bug in non-blocking and EDMA transfer APIs that kStatus_InvalidArgument is returned if user configures 3-wire mode and full-duplex transfer at the same time, but transfer state is already set to kLPSPI_Busy by mistake causing following transfer can not start.
 - Fixed bug when LPSPI slave using EDMA way to transfer, tx should be masked when tx data is null, otherwise in 3-wire mode which tx/rx use the same pin, the received data will be interfered.

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed the bug that the clock source should be divided by the PRESCALE setting in LPSPI_MasterSetDelayTimes function.
 - Fixed the bug that LPSPI_MasterTransferBlocking function would hang in some corner cases.
- Optimization
 - Added #ifndef/#endif to allow user to change the default TX value at compile time.

[2.0.0]

- Initial version.
-

LPSPI_EDMA**[2.4.9]**

- Improvements
 - Removed unused code from LPSPI_SeparateEdmaReadData().

[2.4.8]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

- Improvements
 - Add the TXMSK wait after TCR setting.

[2.4.0]

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

LPUART

[2.10.0]

- New Feature
 - Added support to configure RTS watermark.

[2.9.4]

- Improvements
 - Merged duplicate code.

[2.9.3]

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

[2.9.2]

- Bug Fixes
 - Fixed coverity issues.

[2.9.1]

- Bug Fixes
 - Fixed coverity issues.

[2.9.0]

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

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.7]

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

[2.7.6]

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

[2.7.5]

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

[2.7.4]

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

[2.7.3]

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

[2.7.2]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

- Improvements
 - Use separate data for TX and RX in lpuart_transfer_t.
- Bug Fixes

- Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling LPUART_TransferReceiveNonBlocking, the received data count returned by LPUART_TransferGetReceiveCount is wrong.

[2.5.0]

- Bug Fixes
 - Added missing interrupt enable masks kLPUART_Match1InterruptEnable and kLPUART_Match2InterruptEnable.
 - Fixed bug in LPUART_EnableInterrupts, LPUART_DisableInterrupts and LPUART_GetEnabledInterrupts that the BAUD[LBKDIE] bit field should be soc specific.
 - Fixed bug in LPUART_TransferHandleIRQ that idle line interrupt should be disabled when rx data size is zero.
 - Deleted unused status flags kLPUART_NoiseErrorInRxDataRegFlag and kLPUART_ParityErrorInRxDataRegFlag, since firstly their function are the same as kLPUART_NoiseErrorFlag and kLPUART_ParityErrorFlag, secondly to obtain them one data word must be read out thus interfering with the receiving process.
 - Fixed bug in LPUART_GetStatusFlags that the STAT[LBKDIF], STAT[MA1F] and STAT[MA2F] should be soc specific.
 - Fixed bug in LPUART_ClearStatusFlags that tx/rx FIFO is reset by mistake when clearing flags.
 - Fixed bug in LPUART_TransferHandleIRQ that while clearing idle line flag the other bits should be masked in case other status bits be cleared by accident.
 - Fixed bug of race condition during LPUART transfer using transactional APIs, by disabling and re-enabling the global interrupt before and after critical operations on interrupt enable register.
 - Fixed DMA/eDMA transfer blocking issue by enabling tx idle interrupt after DMA/eDMA transmission finishes.
- New Features
 - Added APIs LPUART_GetRxFifoCount/LPUART_GetTxFifoCount to get rx/tx FIFO data count.
 - Added APIs LPUART_SetRxFifoWatermark/LPUART_SetTxFifoWatermark to set rx/tx FIFO water mark.

[2.4.1]

- Bug Fixes
 - Fixed MISRA advisory 17.7 issues.

[2.4.0]

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

[2.3.1]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.3.0]

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

[2.2.8]

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

[2.2.7]

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

[2.2.6]

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

[2.2.5]

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

[2.2.4]

- Improvements
 - Added hardware flow control function support.
 - Added idle-line-detecting feature in LPUART_TransferNonBlocking function. If an idle line is detected, a callback is triggered with status `kStatus_LPUART_IdleLineDetected` returned. This feature may be useful when the received Bytes is less than the expected received data size. Before triggering the callback, data in the FIFO (if has FIFO) is read out, and no interrupt will be disabled, except for that the receive data size reaches 0.
 - Enabled the RX FIFO watermark function. With the idle-line-detecting feature enabled, users can set the watermark value to whatever you want (should be less than the RX FIFO size). Data is received and a callback will be triggered when data receive ends.

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - Update transactional APIs.
-

LPUART_EDMA

[2.4.0]

- Refer LPUART driver change log 2.1.0 to 2.4.0
-

OCOTP

[2.1.4]

- Bug fixes
 - Fixed the bug that OCOTP_ReadFuseShadowRegisterExt can't read more than one word.

[2.1.3]

- Bug fixes
 - Fixed MISRA 2012 issue: 8.4, 10.3, 10.4, 14.3.
 - Fixed doxygen warning.

[2.1.2]

- Improvements
 - Updated for new MIMXRT117X header file.

[2.1.1]

- Improvements
 - Updated OCOTP_ReloadShadowRegister to return error status.
 - Added functions OCOTP_ReadFuseShadowRegisterExt and OCOTP_WriteFuseShadowRegisterWithLock.
- Bug fixes
 - Fixed MISRA 2012 rule 10.3 issue.

[2.0.1]

- Bug Fixes
 - Fixed doxygen issues.

[2.0.0]

- Initial version.
-

OTFAD**[2.1.4]**

- Bug fixes
 - Fixed MISRA 2012 issue: 10.1.

[2.1.3]

- Bug fixes
 - Fixed the error that waiting for both FLEXSPI AHB idle and SEQ idle.

[2.1.2]

- Bug fixes
 - Fixed MISRA 2012 issue: 10.4.

[2.1.1]

- Improvements:
 - Hided some bits in CR and SR registers for selected platforms.
 - Fixed doxygen issues.

[2.1.0]

- Improvements:
 - Used boolean type to define 1-bit field concepts.

[2.0.0]

- Initial version.
-

PIT**[2.2.0]**

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

- Ensure PIT peripheral clock enabled in PIT_RTI_Init.
- New Features
 - Added PIT_ClearRtiSyncStatus API to clear the RTI_LDVAL_STAT register.

[2.1.1]

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

[2.1.0]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

PMU**[2.1.1]**

- Bug Fixes
 - Fixed the violations of MISRA 2012 rules: Rule 10.1 10.4

[2.1.0]

- Improvements
 - Added feature macros for low power control APIs to support conditional compile.
 - Renamed “PMU_2P1EnablePullDown” to “PMU_2P5EnablePullDown”.

[2.0.0]

- Initial version.
-

PWM**[2.9.1]**

- Improvements
 - Add new API PWM_SetupFaultsExt and PWM_SetupFaultInputFilterExt to support Flex-PWM which has more than one fault input channels.
 - Support fault 4-7 interrupt and its flag.
- Bug Fixes
 - Fixed violations of the CERT INT31-C.

[2.9.0]

- Improvements
 - Support PWMX channel output for edge aligned PWM.
 - Forbid submodule 0 counter initialize with master sync and master reload mode.
 - Clarify kPWM_BusClock meaning.
 - Verify pulseCnt within 65535 when update period register.

[2.8.4]

- Improvements
 - Support workaround for ERR051989. This function helps realize no phase delay between submodule 0 and other submodule.

[2.8.3]

- Bug Fixes
 - Fixed MISRA C-2012 Rule 15.7

[2.8.2]

- Bug Fixes
 - Fixed warning conversion from ‘int’ to ‘uint16_t’ on API PWM_Init.
 - Fixed warning unused variable ‘reg’ on API PWM_SetPwmForceOutputToZero.

[2.8.1]

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

[2.8.0]

- Improvements
 - Added API PWM_UpdatePwmPeriodAndDutycycle to update the PWM signal’s period and dutycycle for a PWM submodule.
 - Added API PWM_SetPeriodRegister and PWM_SetDutycycleRegister to merge duplicate code in API PWM_SetupPwm, PWM_UpdatePwmDutycycleHighAccuracy and PWM_UpdatePwmPeriodAndDutycycle

[2.7.1]

- Improvements
 - Supported UPDATE_MASK bit in MASK register.

[2.7.0]

- Improvements
 - Supported platforms which don’t have Capture feature with channel A and B.
 - Supported platforms which don’t have Submodule 3.
 - Added assert function in API PWM_SetPhaseDelay to prevent wrong argument.

[2.6.1]

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

[2.6.0]

- Improvements
 - Added API PWM_SetPhaseDelay to set the phase delay from the master sync signal of submodule 0.
 - Added API PWM_SetFilterSampleCount to set number of consecutive samples that must agree prior to the input filter.
 - Added API PWM_SetFilterSamplePeriod to set the sampling period of the fault pin input filter.

[2.5.1]

- Bug Fixes
 - Fixed MISRA C-2012 rules: 10.1, 10.3, 10.4 , 10.6 and 10.8.
 - Fixed the issue that PWM_UpdatePwmDutycycle() can't update duty cycle status value correct.

[2.5.0]

- Improvements
 - Added API PWM_SetOutputToIdle to set pwm channel output to idle.
 - Added API PWM_GetPwmChannelState to get the pwm channel output duty cycle value.
 - Added API PWM_SetPwmForceOutputToZero to set the pwm channel output to zero logic.
 - Added API PWM_SetChannelOutput to set the pwm channel output state.
 - Added API PWM_SetClockMode to set the value of the clock prescaler.
 - Added API PWM_SetupPwmPhaseShift to set PWM which a special phase shift and 50% duty cycle.
 - Added API PWM_SetVALxValue/PWM_GetVALxValue to set/get PWM VALs registers values directly.

[2.4.0]

- Improvements
 - Supported the PWM which can't work in wait mode.

[2.3.0]

- Improvements
 - Add PWM output enable&disbale API for SDK.
- Bug Fixes
 - Fixed changing channel B configuration when parameter is kPWM_PWMX and PWMX configuration is not supported yet.

[2.2.1]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rules: 10.3, 10.4.
- Bug Fixes
 - Fixed the issue that PWM drivers computed VAL1 improperly.
- Improvements
 - Updated calculation accuracy of reloadValue in dutyCycleToReloadValue function.

[2.2.0]

- Improvements
 - Added new enumeration and two APIs to support enabling and disabling one or more PWM output triggers.
 - Added a new function to make the most of 16-bit resolution PWM.
 - Added one API to support updating fault status of PWM output.
 - Added one API to support PWM DMA write request.
 - Added three APIs to support PWM DMA capture read request.
 - Added one API to support get default fault config of PWM.
 - Added one API to support setting PWM fault disable mapping.

[2.1.0]

- Improvements
 - Moved the configuration of fault input filter into a new API to avoid be initialized multiple times.
- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fix rules, containing: rule-10.2, rule-10.3, rule-10.4, rule-10.7, rule-10.8, rule-14.4, rule-16.4.

[2.0.1]

- Bug Fixes
 - Fixed the issue that PWM submodule may be initialized twice in function PWM_SetupPwm().

[2.0.0]

- Initial version.
-

RTWDOG

[2.1.4]

- Bug Fixes
 - Fixed CERT INT30-C, INT31-C issue.
 - Make API RTWDOG_CountToMesec return 0 if result overflow.

[2.1.3]

- Improvements
 - Waited the over status after CS register operation in case next CS operation causes problem.

[2.1.2]

- Bug Fixes
 - Fixed doxygen issue.

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fixed rules, containing: rule-10.3, rule-10.8, rule-11.9, rule-14.4, rule-15.5.

[2.1.0]

- Improvements
 - Added an API to enable or disable the window mode.
 - Added an API to convert a raw count value to millisecond.
 - Used AT_QUICKACCESS_SECTION_CODE macro to decorate RTWDOG_Init, and copied this function from flash to QUICKACCESS section.

[2.0.1]

- Bug Fixes
 - Fixed bug in the RTWDOG_Init; added check for register's unlock status when configuring the RTWDOG in RTWDOG_init.

[2.0.0]

- Initial version.
-

SAI

[2.4.10]

- Improvements
 - Allow enabling/disabling implicit channel configuration.
 - Allow NULL FIFO watermark.
- Bug Fixes
 - Fix compilation warnings when asserts are disabled

[2.4.9]

- Added Errata ERR051421 workaround.

[2.4.8]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 12.4.

[2.4.7]

- Added conditional support for bit clock swap feature
- Added common IRQ handler entry SAI_DriverIRQHandler.

[2.4.6]

- Bug Fixes
 - Fixed the IAR build warning.

[2.4.5]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 12.4.

[2.4.4]

- Bug Fixes
 - Fixed enumeration sai_fifo_combine_t - add RX configuration.

[2.4.3]

- Bug Fixes
 - Fixed enumeration sai_fifo_combine_t value configuration issue.

[2.4.2]

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

[2.4.1]

- Bug Fixes
 - Fixed bitWidth incorrectly assigned issue.

[2.4.0]

- Improvements
 - Removed deprecated APIs.

[2.3.8]

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

[2.3.7]

- Improvements
 - Change feature “FSL_FEATURE_SAI_FIFO_COUNT” to “FSL_FEATURE_SAI_HAS_FIFO”.
 - Added feature “FSL_FEATURE_SAI_FIFO_COUNTn(x)” to align SAI fifo count function with IP in function

[2.3.6]

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

[2.3.5]

- Improvements
 - Make driver to be aarch64 compatible.

[2.3.4]

- Bug Fixes
 - Corrected the fifo combine feature macro used in driver.

[2.3.3]

- Bug Fixes
 - Added bit clock polarity configuration when sai act as slave.
 - Fixed out of bound access coverity issue.
 - Fixed violations of MISRA C-2012 rule 10.3, 10.4.

[2.3.2]

- Bug Fixes
 - Corrected the frame sync configuration when sai act as slave.

[2.3.1]

- Bug Fixes
 - Corrected the peripheral name in function SAI0_DriverIRQHandler.
 - Fixed violations of MISRA C-2012 rule 17.7.

[2.3.0]

- Bug Fixes
 - Fixed the build error caused by the SOC has no fifo feature.

[2.2.3]

- Bug Fixes
 - Corrected the peripheral name in function SAI0_DriverIRQHandler.

[2.2.2]

- Bug Fixes
 - Fixed the issue of MISRA 2004 rule 9.3.
 - Fixed sign-compare warning.
 - Fixed the PA082 build warning.
 - Fixed sign-compare warning.
 - Fixed violations of MISRA C-2012 rule 10.3,17.7,10.4,8.4,10.7,10.8,14.4,17.7,11.6,10.1,10.6,8.4,14.3,16.4,18.2.
 - Allow to reset Rx or Tx FIFO pointers only when Rx or Tx is disabled.
- Improvements
 - Added 24bit raw audio data width support in sai sdma driver.
 - Disabled the interrupt/DMA request in the SAI_Init to avoid generates unexpected sai FIFO requests.

[2.2.1]

- Improvements
 - Added mclk post divider support in function SAI_SetMasterClockDivider.
 - Removed useless configuration code in SAI_RxSetSerialDataConfig.
- Bug Fixes
 - Fixed the SAI SDMA driver build issue caused by the wrong structure member name used in the function SAI_TransferRxSetConfigSDMA/SAI_TransferTxSetConfigSDMA.
 - Fixed BAD BIT SHIFT OPERATION issue caused by the FSL_FEATURE_SAI_CHANNEL_COUNTn.
 - Applied ERR05144: not set FCONT = 1 when TMR > 0, otherwise the TX may not work.

[2.2.0]

- Improvements
 - Added new APIs for parameters collection and simplified user interfaces:
 - * SAI_Init
 - * SAI_SetMasterClockConfig
 - * SAI_TxSetBitClockRate
 - * SAI_TxSetSerialDataConfig
 - * SAI_TxSetFrameSyncConfig
 - * SAI_TxSetFifoConfig
 - * SAI_TxSetBitclockConfig
 - * SAI_TxSetConfig
 - * SAI_TxSetTransferConfig
 - * SAI_RxSetBitClockRate
 - * SAI_RxSetSerialDataConfig
 - * SAI_RxSetFrameSyncConfig
 - * SAI_RxSetFifoConfig
 - * SAI_RxSetBitclockConfig
 - * SAI_RXSetConfig
 - * SAI_RxSetTransferConfig
 - * SAI_GetClassicI2SConfig
 - * SAI_GetLeftJustifiedConfig
 - * SAI_GetRightJustifiedConfig
 - * SAI_GetTDMConfig

[2.1.9]

- Improvements
 - Improved SAI driver comment for clock polarity.
 - Added enumeration for SAI for sample inputs on different edges.
 - Changed FSL_FEATURE_SAI_CHANNEL_COUNT to FSL_FEATURE_SAI_CHANNEL_COUNTn(base) for the difference between the different SAI instances.
- Added new APIs:
 - SAI_TxSetBitClockDirection
 - SAI_RxSetBitClockDirection
 - SAI_RxSetFrameSyncDirection
 - SAI_TxSetFrameSyncDirection

[2.1.8]

- Improvements
 - Added feature macro test for the sync mode2 and mode 3.
 - Added feature macro test for masterClockHz in sai_transfer_format_t.

[2.1.7]

- Improvements
 - Added feature macro test for the mclkSource member in sai_config_t.
 - Changed “FSL_FEATURE_SAI5_SAI6_SHARE_IRQ” to “FSL_FEATURE_SAI_SAI5_SAI6_SHARE_IRQ”.
 - Added #ifndef #endif check for SAI_XFER_QUEUE_SIZE to allow redefinition.
- Bug Fixes
 - Fixed build error caused by feature macro test for mclkSource.

[2.1.6]

- Improvements
 - Added feature macro test for mclkSourceClockHz check.
 - Added bit clock source name for general devices.
- Bug Fixes
 - Fixed incorrect channel numbers setting while calling RX/TX set format together.

[2.1.5]

- Bug Fixes
 - Corrected SAI3 driver IRQ handler name.
 - Added I2S4/5/6 IRQ handler.
 - Added base in handler structure to support different instances sharing one IRQ number.
- New Features
 - Updated SAI driver for MCR bit MICS.
 - Added 192 KHZ/384 KHZ in the sample rate enumeration.
 - Added multi FIFO interrupt/SDMA transfer support for TX/RX.
 - Added an API to read/write multi FIFO data in a blocking method.
 - Added bclk bypass support when bclk is same with mclk.

[2.1.4]

- New Features
 - Added an API to enable/disable auto FIFO error recovery in platforms that support this feature.
 - Added an API to set data packing feature in platforms which support this feature.

[2.1.3]

- New Features
 - Added feature to make I2S frame sync length configurable according to bitWidth.

[2.1.2]

- Bug Fixes
 - Added 24-bit support for SAI eDMA transfer. All data shall be 32 bits for send/receive, as eDMA cannot directly handle 3-Byte transfer.

[2.1.1]

- Improvements
 - Reduced code size while not using transactional API.

[2.1.0]

- Improvements
 - API name changes:
 - * SAI_GetSendRemainingBytes -> SAI_GetSentCount.
 - * SAI_GetReceiveRemainingBytes -> SAI_GetReceivedCount.
 - * All names of transactional APIs were added with “Transfer” prefix.
 - * All transactional APIs use base and handle as input parameter.
 - * Unified the parameter names.
- Bug Fixes
 - Fixed WLC bug while reading TCSR/RCSR registers.
 - Fixed MOE enable flow issue. Moved MOE enable after MICS settings in SAI_TxInit/SAI_RxInit.

[2.0.0]

- Initial version.
-

SAI_EDMA

[2.7.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 12.4.

[2.7.2]

- Improvements
 - Add macros `MCUX_SDK_SAI_EDMA_TX_ENABLE_INTERNAL` and `MCUX_SDK_SAI_EDMA_RX_ENABLE_INTERNAL` to let the user decide whether to enable SAI when calling `SAI_TransferSendEDMA/SAI_TransferReceiveEDMA`.

[2.7.1]

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

[2.7.0]

- Improvements
 - Updated api SAI_TransferReceiveEDMA to support voice channel block interleave transfer.
 - Updated api SAI_TransferSendEDMA to support voice channel block interleave transfer.
 - Added new api SAI_TransferSetInterleaveType to support channel interleave type configurations.

[2.6.0]

- Improvements
 - Removed deprecated APIs.

[2.5.1]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 20.7.

[2.5.0]

- Improvements
 - Added new api SAI_TransferSendLoopEDMA/SAI_TransferReceiveLoopEDMA to support loop transfer.
 - Added multi sai channel transfer support.

[2.4.0]

- Improvements
 - Added new api SAI_TransferGetValidTransferSlotsEDMA which can be used to get valid transfer slot count in the sai edma transfer queue.
 - Deprecated the api SAI_TransferRxSetFormatEDMA and SAI_TransferTxSetFormatEDMA.
- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.3,10.4.

[2.3.2]

- Refer SAI driver change log 2.1.0 to 2.3.2

SNVS_HP

[2.3.2]

- Make SNVS_HP_RTC_Init()/SNVS_HP_RTC_Deinit more transparent. Use function SNVS_HP_Init()/SNVS_HP_Deinit() instead of copy of this code in SNVS_HP_RTC_XXX() function.

[2.3.1]

- Fixed problem in SNVS_HP_RTC_Init(), which is clearing bits that should stay intact.

[2.3.0]

- Re-map Security Violation for RT11xx specific violations.

[2.2.0]

- Fixed doxygen issues.
- Add SNVS HP Set locks.

[2.1.4]

- Fix MISRA issues.

[2.1.3]

- Fixed IAR Pa082 warnings.

[2.1.2]

- Fixed problem with initialization of the periodic interrupt frequency.
- Fixed problem with SNVS entering into fail state when HAB enters closed mode.

[2.1.1]

- Added APIs for HP security violation status flags.

[2.1.0]

- Added APIs for High Assurance Counter (HAC), Zeroizable Master Key (ZMK) and Software Security Violation.

[2.0.0]

- Initial version.
-

SNVS_LP

[2.4.6]

- Fix a bug in SNVS_LP_EnableRxActiveTamper() where assignments to base->LPATRC2R were done wrongly to LPATRC1R.

[2.4.5]

- Fix a bug in SNVS_LP_EnableRxActiveTamper() where assignments to base->LPATRC1R would overwrite previously set bits.

[2.4.4]

- Make SNVS_LP_SRTC_Init()/SNVS_LP_SRTC_Deinit more transparent. Use function SNVS_LP_Init()/SNVS_LP_Deinit() instead of copy of this code in SNVS_LP_SRTC_XXX() function.

[2.4.3]

- Fixed problem in SNVS_LP_SRTC_Init(), which is clearing bits that should stay intact.

[2.4.2]

- Updated driver to match with new device header files.

[2.4.1]

- Fixed MISRA issues.

[2.4.0]

- Fix backward compatibility with version 2.2.x.

[2.3.0]

- Add active pin, clock, voltage and temperature tamper features.

[2.2.0]

- Fixed doxygen issues.
- Add Transition SNVS SSM state to Trusted/Non-secure from Check state.

[2.1.2]

- Fix MISRA issues.

[2.1.1]

- Fix IAR Pa082 warning.

[2.1.0]

- Added APIs for Zeroizable Master Key (ZMK) and Monotonic Counter (MC).

[2.0.0]

- Initial version.
-

SPDIF**[2.0.7]**

- Improvements
 - Add feature macro `FSL_FEATURE_SPDIF_HAS_NO_SIC_REGISTER` to handle non-existent SIC register.

[2.0.6]

- Bug Fixes
 - Fixed the Q/U channel interrupt enabled unexpectedly while Q/U transfer pointer is NULL.

[2.0.5]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 11.3.

[2.0.4]

- Bug Fixes
 - Added udata/qdata buffer address validation in driver IRQ handler to ensure that NULL pointer dereferences do not occur.

[2.0.3]

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

[2.0.2]

- Bug Fixes
 - Corrected operator used for size value assertion in `SPDIF_ReadBlocking/SPDIF_WriteBlocking`.

[2.0.1]

- Bug Fixes
 - Corrected the feature macro name used to define `s_edmaPrivateHandle`.

[2.0.0]

- Initial version.
-

SPDIF DMA Driver

[2.0.8]

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

[2.0.7]

- Bug Fixes
 - Fixed the incompatibility issue with edma4 driver.

[2.0.6]

- Bug Fixes
 - Add feature macro to determine whether to use the API MEMORY_ConvertMemoryMapAddress to translate TCD addresses for DLAST_SGA.

[2.0.5]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 11.3.

[2.0.4]

- Bug Fixes
 - Added udata/qdata buffer address validation in driver IRQ handler to ensure that NULL pointer dereferences do not occur.

[2.0.3]

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

[2.0.2]

- Bug Fixes
 - Corrected operator used for size value assertion in SPDIF_ReadBlocking/SPDIF_WriteBlocking.

[2.0.1]

- Bug Fixes
 - Corrected the feature macro name used to define s_edmaPrivateHandle.

[2.0.0]

- Initial version.
-

SRC**[2.0.1]**

- Improvements
 - Updated SRC driver for adding SRC_SRSR_JTAG_SW_RST enumeration.

[2.0.0]

- Initial version.
-

TEMPMON**[2.2.1]**

- Improvements
 - Move TEMPMON calibration data mask definitions from the driver source file to the header file, as the driver examples also need to use these definitions.

[2.2.0]

- Bug Fixes
 - Fixed the issue of inconsistency between data width and RM.

[2.1.1]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.3 10.4.

[2.1.0]

- Bug Fixes
 - Supported minus value for alarm temperature setting.
 - Fixed wrong temperature calculation equation.

[2.0.3]

- Improvements
 - Added temperature threshold check for high/low/panic to avoid temperature overflow.

[2.0.2]

- Bug Fixes
 - Fixed wrong alarm value setting API, it need to clear it firstly and set a new value into it.

[2.0.1]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.1 10.3 10.4 10.8 17.7.

[2.0.0]

- Initial version.
-

TRNG

[2.0.19]

- New features:
 - Added support for MCXA and MCXL.

[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.
-

WDOG

[2.2.0]

- Bug Fixes
 - Fixed the wrong behavior of workMode.enableWait, workMode.enableStop, workMode.enableDebug in configuration structure wdog_config_t. When set the items to true, WDOG will continues working in those modes.

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.1, 10.3, 10.4, 10.6, 10.7 and 11.9.
 - Fixed the issue of the inseparable process interrupted by other interrupt source.
 - * WDOG_Init
 - * WDOG_Refresh

[2.1.0]

- New Features
 - Added new API “WDOG_TriggerSystemSoftwareReset()” to allow users to reset the system by software.
 - Added new API “WDOG_TriggerSoftwareSignal()” to allow users to trigger a WDOG_B signal by software.
 - Removed the parameter “softwareAssertion” and “softwareResetSignal” out of the wdog_config_t structure.
 - Added new parameter “enableTimeOutAssert” to the wdog_config_t structure. With this parameter enabled, when the WDOG timeout occurs, a WDOG_B signal will be asserted. This signal can be routed to external pin of the chip. Note that WDOG_B signal remains asserted until a power-on reset (POR) occurs.

[2.0.1]

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

[2.0.0]

- Initial version.
-

XBARA

[2.0.6]

- Bug Fixes
 - Fixed typo in kXBARA_RequestInterruptEnalbe item.

[2.0.5]

- Bug Fixes
 - Fixed IAR build warning Pa082.
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.7, 10.8, 12.1, 18.1, 20.7.

[2.0.4]

- Improvements
 - Optimized XBARA_SetOutputSignalConfig.

[2.0.3]

- Bug Fixes
 - Corrected configuration for function XBAR_SetOutputSignalConfig.

[2.0.2]

- Other Changes
 - Changed array clock name.

[2.0.1]

- Bug Fixes
 - Fixed w1c bits for XBARA_SetOutputSignalConfig function.

[2.0.0]

- Initial version.
-

1.6 Driver API Reference Manual

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

[MIMXRT1011](#)

1.7 Middleware Documentation

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

1.7.1 FreeMASTER

[freemaster](#)

1.7.2 FreeRTOS

[FreeRTOS](#)

1.7.3 File systemFatfs

fatfs

Chapter 2

MIMXRT1011

2.1 ADC: 12-bit Analog to Digital Converter Driver

`void ADC_Init(ADC_Type *base, const adc_config_t *config)`

Initialize the ADC module.

Parameters

- `base` – ADC peripheral base address.
- `config` – Pointer to “*adc_config_t*” structure.

`void ADC_Deinit(ADC_Type *base)`

De-initializes the ADC module.

Parameters

- `base` – ADC peripheral base address.

`void ADC_GetDefaultConfig(adc_config_t *config)`

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

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

```
config->enableAsynchronousClockOutput = true;
config->enableOverWrite = false;
config->enableContinuousConversion = false;
config->enableHighSpeed = false;
config->enableLowPower = false;
config->enableLongSample = false;
config->referenceVoltageSource = kADC_ReferenceVoltageSourceAlt0;
config->samplePeriodMode = kADC_SamplePeriod2or12Clocks;
config->clockSource = kADC_ClockSourceAD;
config->clockDriver = kADC_ClockDriver1;
config->resolution = kADC_Resolution12Bit;
```

Parameters

- `config` – Pointer to the configuration structure.

`void ADC_SetChannelConfig(ADC_Type *base, uint32_t channelGroup, const adc_channel_config_t *config)`

Configures the conversion channel.

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

Note that the “Channel Group” has a detailed description. To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC has more than one group of status and control registers, one for each conversion. The channel group parameter indicates which group of registers are used, for example channel group 0 is for Group A registers and channel group 1 is for Group B registers. The channel groups are used in a “ping-pong” approach to control the ADC operation. At any point, only one of the channel groups is actively controlling ADC conversions. The channel group 0 is used for both software and hardware trigger modes. Channel groups 1 and greater indicate potentially multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual about the number of SC1n registers (channel groups) specific to this device. None of the channel groups 1 or greater are used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

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

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

Gets the conversion value.

Parameters

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

Returns

Conversion value.

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

Gets the status flags of channel.

A conversion is completed when the result of the conversion is transferred into the data result registers. (provided the compare function & hardware averaging is disabled), this is indicated by the setting of COCON. If hardware averaging is enabled, COCON sets only, if the last of the selected number of conversions is complete. If the compare function is enabled, COCON sets and conversion result data is transferred only if the compare condition is true. If both hardware averaging and compare functions are enabled, then COCON sets only if the last of the selected number of conversions is complete and the compare condition is true.

Parameters

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

Returns

Status flags of channel. return 0 means COCO flag is 0, return 1 means COCO flag is 1.

status_t ADC_DoAutoCalibration(ADC_Type *base)

Automates the hardware calibration.

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

Parameters

- base – ADC peripheral base address.

Return values

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

Returns

Execution status.

void ADC_SetOffsetConfig(ADC_Type *base, const *adc_offset_config_t* *config)

Set user defined offset.

Parameters

- base – ADC peripheral base address.
- config – Pointer to “*adc_offset_config_t*” structure.

static inline void ADC_EnableDMA(ADC_Type *base, bool enable)

Enables generating the DMA trigger when the conversion is complete.

Parameters

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

void ADC_SetHardwareCompareConfig(ADC_Type *base, const *adc_hardware_compare_config_t* *config)

Enables the hardware trigger mode.

Configures the hardware compare mode.

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

Parameters

- base – ADC peripheral base address.
- enable – Switcher of the trigger mode. “true” means hardware trigger mode, “false” means software mode.
- base – ADC peripheral base address.
- config – Pointer to “*adc_hardware_compare_config_t*” structure.

void ADC_SetHardwareAverageConfig(ADC_Type *base, *adc_hardware_average_mode_t* mode)

Configures the hardware average mode.

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

Parameters

- base – ADC peripheral base address.
- mode – Setting the hardware average mode. See “adc_hardware_average_mode_t”.

static inline uint32_t ADC_GetStatusFlags(ADC_Type *base)

Gets the converter’s status flags.

Parameters

- base – ADC peripheral base address.

Returns

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

void ADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clears the converter’s status falgs.

Parameters

- base – ADC peripheral base address.
- mask – Mask value for the cleared flags. See “adc_status_flags_t”.

enum _adc_status_flags

Converter’s status flags.

Values:

enumerator kADC_ConversionActiveFlag

Conversion is active,not support w1c.

enumerator kADC_CalibrationFailedFlag

Calibration is failed,support w1c.

enumerator kADC_AsynchronousWakeupInterruptFlag

Asynchronous wakeup interrupt occurred, support w1c.

enum _adc_reference_voltage_source

Reference voltage source.

Values:

enumerator kADC_ReferenceVoltageSourceAlt0

For external pins pair of VrefH and VrefL.

enum _adc_sample_period_mode

Sample time duration.

Values:

enumerator kADC_SamplePeriod2or12Clocks

Long sample 12 clocks or short sample 2 clocks.

enumerator kADC_SamplePeriod4or16Clocks

Long sample 16 clocks or short sample 4 clocks.

enumerator kADC_SamplePeriod6or20Clocks

Long sample 20 clocks or short sample 6 clocks.

enumerator kADC_SamplePeriod8or24Clocks

Long sample 24 clocks or short sample 8 clocks.

enumerator kADC_SamplePeriodLong12Clcoks

Long sample 12 clocks.

enumerator kADC__SamplePeriodLong16Clcoks

Long sample 16 clocks.

enumerator kADC__SamplePeriodLong20Clcoks

Long sample 20 clocks.

enumerator kADC__SamplePeriodLong24Clcoks

Long sample 24 clocks.

enumerator kADC__SamplePeriodShort2Clocks

Short sample 2 clocks.

enumerator kADC__SamplePeriodShort4Clocks

Short sample 4 clocks.

enumerator kADC__SamplePeriodShort6Clocks

Short sample 6 clocks.

enumerator kADC__SamplePeriodShort8Clocks

Short sample 8 clocks.

enum _adc_clock_source

Clock source.

Values:

enumerator kADC__ClockSourceIPG

Select IPG clock to generate ADCK.

enumerator kADC__ClockSourceIPGDiv2

Select IPG clock divided by 2 to generate ADCK.

enumerator kADC__ClockSourceAD

Select Asynchronous clock to generate ADCK.

enum _adc_clock_drvier

Clock divider for the converter.

Values:

enumerator kADC__ClockDriver1

For divider 1 from the input clock to the module.

enumerator kADC__ClockDriver2

For divider 2 from the input clock to the module.

enumerator kADC__ClockDriver4

For divider 4 from the input clock to the module.

enumerator kADC__ClockDriver8

For divider 8 from the input clock to the module.

enum _adc_resolution

Converter's resolution.

Values:

enumerator kADC__Resolution8Bit

Single End 8-bit resolution.

enumerator kADC__Resolution10Bit

Single End 10-bit resolution.

enumerator `kADC_Resolution12Bit`

Single End 12-bit resolution.

enum `_adc_hardware_compare_mode`

Converter hardware compare mode.

Values:

enumerator `kADC_HardwareCompareMode0`

Compare true if the result is less than the value1.

enumerator `kADC_HardwareCompareMode1`

Compare true if the result is greater than or equal to value1.

enumerator `kADC_HardwareCompareMode2`

Value1 <= Value2, compare true if the result is less than value1 Or the result is Greater than value2. Value1 > Value2, compare true if the result is less than value1 And the result is greater than value2

enumerator `kADC_HardwareCompareMode3`

Value1 <= Value2, compare true if the result is greater than or equal to value1 And the result is less than or equal to value2. Value1 > Value2, compare true if the result is greater than or equal to value1 Or the result is less than or equal to value2.

enum `_adc_hardware_average_mode`

Converter hardware average mode.

Values:

enumerator `kADC_HardwareAverageCount4`

For hardware average with 4 samples.

enumerator `kADC_HardwareAverageCount8`

For hardware average with 8 samples.

enumerator `kADC_HardwareAverageCount16`

For hardware average with 16 samples.

enumerator `kADC_HardwareAverageCount32`

For hardware average with 32 samples.

enumerator `kADC_HardwareAverageDiasable`

Disable the hardware average function.

typedef enum `_adc_status_flags` `adc_status_flags_t`

Converter's status flags.

typedef enum `_adc_reference_voltage_source` `adc_reference_voltage_source_t`

Reference voltage source.

typedef enum `_adc_sample_period_mode` `adc_sample_period_mode_t`

Sample time duration.

typedef enum `_adc_clock_source` `adc_clock_source_t`

Clock source.

typedef enum `_adc_clock_drvier` `adc_clock_driver_t`

Clock divider for the converter.

typedef enum `_adc_resolution` `adc_resolution_t`

Converter's resolution.

`typedef enum _adc_hardware_compare_mode` `adc_hardware_compare_mode_t`
 Converter hardware compare mode.

`typedef enum _adc_hardware_average_mode` `adc_hardware_average_mode_t`
 Converter hardware average mode.

`typedef struct _adc_config` `adc_config_t`
 Converter configuration.

`typedef struct _adc_offset_config` `adc_offset_config_t`
 Converter Offset configuration.

`typedef struct _adc_hardware_compare_config` `adc_hardware_compare_config_t`
 ADC hardware compare configuration.

In `kADC_HardwareCompareMode0`, compare true if the result is less than the value1. In `kADC_HardwareCompareMode1`, compare true if the result is greater than or equal to value1. In `kADC_HardwareCompareMode2`, Value1 <= Value2, compare true if the result is less than value1 Or the result is Greater than value2. Value1 > Value2, compare true if the result is less than value1 And the result is Greater than value2. In `kADC_HardwareCompareMode3`, Value1 <= Value2, compare true if the result is greater than or equal to value1 And the result is less than or equal to value2. Value1 > Value2, compare true if the result is greater than or equal to value1 Or the result is less than or equal to value2.

`typedef struct _adc_channel_config` `adc_channel_config_t`
 ADC channel conversion configuration.

`FSL_ADC_DRIVER_VERSION`
 ADC driver version.
 Version 2.0.4.

`struct _adc_config`
`#include <fsl_adc.h>` Converter configuration.

Public Members

`bool enableOverWrite`
 Enable the overwriting.

`bool enableContinuousConversion`
 Enable the continuous conversion mode.

`bool enableHighSpeed`
 Enable the high-speed mode.

`bool enableLowPower`
 Enable the low power mode.

`bool enableLongSample`
 Enable the long sample mode.

`bool enableAsynchronousClockOutput`
 Enable the asynchronous clock output.

`adc_reference_voltage_source_t` `referenceVoltageSource`
 Select the reference voltage source.

`adc_sample_period_mode_t` `samplePeriodMode`
 Select the sample period in long sample mode or short mode.

adc_clock_source_t clockSource

Select the input clock source to generate the internal clock ADCK.

adc_clock_driver_t clockDriver

Select the divide ratio used by the ADC to generate the internal clock ADCK.

adc_resolution_t resolution

Select the ADC resolution mode.

struct *_adc_offset_config*

#include <fsl_adc.h> Converter Offset configuration.

Public Members

bool enableSigned

if false, The offset value is added with the raw result. if true, The offset value is subtracted from the raw converted value.

uint32_t offsetValue

User configurable offset value(0-4095).

struct *_adc_hardware_compare_config*

#include <fsl_adc.h> ADC hardware compare configuration.

In kADC_HardwareCompareMode0, compare true if the result is less than the value1. In kADC_HardwareCompareMode1, compare true if the result is greater than or equal to value1. In kADC_HardwareCompareMode2, Value1 <= Value2, compare true if the result is less than value1 Or the result is Greater than value2. Value1 > Value2, compare true if the result is less than value1 And the result is Greater than value2. In kADC_HardwareCompareMode3, Value1 <= Value2, compare true if the result is greater than or equal to value1 And the result is less than or equal to value2. Value1 > Value2, compare true if the result is greater than or equal to value1 Or the result is less than or equal to value2.

Public Members

adc_hardware_compare_mode_t hardwareCompareMode

Select the hardware compare mode. See “*adc_hardware_compare_mode_t*”.

uint16_t value1

Setting value1(0-4095) for hardware compare mode.

uint16_t value2

Setting value2(0-4095) for hardware compare mode.

struct *_adc_channel_config*

#include <fsl_adc.h> ADC channel conversion configuration.

Public Members

uint32_t channelNumber

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

bool enableInterruptOnConversionCompleted

Generate an interrupt request once the conversion is completed.

2.2 ADC_ETC: ADC External Trigger Control

void ADC_ETC_Init(ADC_ETC_Type *base, const *adc_etc_config_t* *config)

Initialize the ADC_ETC module.

Parameters

- base – ADC_ETC peripheral base address.
- config – Pointer to “*adc_etc_config_t*” structure.

void ADC_ETC_Deinit(ADC_ETC_Type *base)

De-Initialize the ADC_ETC module.

Parameters

- base – ADC_ETC peripheral base address.

void ADC_ETC_GetDefaultConfig(*adc_etc_config_t* *config)

Gets an available pre-defined settings for the ADC_ETC's configuration. This function initializes the ADC_ETC's configuration structure with available settings. The default values are:

```
config->enableTSCBypass = true;
config->enableTSC0Trigger = false;
config->enableTSC1Trigger = false;
config->TSC0triggerPriority = 0U;
config->TSC1triggerPriority = 0U;
config->clockPreDivider = 0U;
config->XBARtriggerMask = 0U;
```

Parameters

- config – Pointer to “*adc_etc_config_t*” structure.

void ADC_ETC_SetTriggerConfig(ADC_ETC_Type *base, uint32_t triggerGroup, const *adc_etc_trigger_config_t* *config)

Set the external XBAR trigger configuration.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index.
- config – Pointer to “*adc_etc_trigger_config_t*” structure.

void ADC_ETC_SetTriggerChainConfig(ADC_ETC_Type *base, uint32_t triggerGroup, uint32_t chainGroup, const *adc_etc_trigger_chain_config_t* *config)

Set the external XBAR trigger chain configuration. For example, if triggerGroup is set to 0U and chainGroup is set to 1U, which means Trigger0 source's chain1 would be configured.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.
- chainGroup – Trigger chain group index. Available number is 0~7.
- config – Pointer to “*adc_etc_trigger_chain_config_t*” structure.

uint32_t ADC_ETC_GetInterruptStatusFlags(ADC_ETC_Type *base, *adc_etc_external_trigger_source_t* sourceIndex)

Gets the interrupt status flags of external XBAR and TSC triggers.

Parameters

- base – ADC_ETC peripheral base address.
- sourceIndex – trigger source index.

Returns

Status flags mask of trigger. Refer to “_adc_etc_status_flag_mask”.

```
void ADC_ETC_ClearInterruptStatusFlags(ADC_ETC_Type *base,  
                                       adc_etc_external_trigger_source_t sourceIndex,  
                                       uint32_t mask)
```

Clears the ADC_ETC's interrupt status falgs.

Parameters

- base – ADC_ETC peripheral base address.
- sourceIndex – trigger source index.
- mask – Status flags mask of trigger. Refer to “_adc_etc_status_flag_mask”.

```
static inline void ADC_ETC_EnableDMA(ADC_ETC_Type *base, uint32_t triggerGroup)
```

Enable the DMA corresponding to each trigger source.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.

```
static inline void ADC_ETC_DisableDMA(ADC_ETC_Type *base, uint32_t triggerGroup)
```

Disable the DMA corresponding to each trigger sources.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.

```
static inline uint32_t ADC_ETC_GetDMAStatusFlags(ADC_ETC_Type *base)
```

Get the DMA request status falgs. Only external XBAR sources support DMA request.

Parameters

- base – ADC_ETC peripheral base address.

Returns

Mask of external XBAR tirgger's DMA request asserted flags. Available range is trigger0:0x01 to trigger7:0x80.

```
static inline void ADC_ETC_ClearDMAStatusFlags(ADC_ETC_Type *base, uint32_t mask)
```

Clear the DMA request status falgs. Only external XBAR sources support DMA request.

Parameters

- base – ADC_ETC peripheral base address.
- mask – Mask of external XBAR tirgger's DMA request asserted flags. Available range is trigger0:0x01 to trigger7:0x80.

```
static inline void ADC_ETC_DoSoftwareReset(ADC_ETC_Type *base, bool enable)
```

When enable, all logical will be reset.

Parameters

- base – ADC_ETC peripheral base address.
- enable – Enable/Disable the software reset.

```
static inline void ADC_ETC_DoSoftwareTrigger(ADC_ETC_Type *base, uint32_t triggerGroup)
```

Do software trigger corresponding to each XBAR trigger sources. Each XBAR trigger sources can be configured as HW or SW trigger mode. In hardware trigger mode, trigger source is from XBAR. In software mode, trigger source is from software trigger. TSC trigger sources can only work in hardware trigger mode.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.

```
static inline void ADC_ETC_DoSoftwareTriggerBlocking(ADC_ETC_Type *base, uint32_t
                                                    triggerGroup)
```

Do software trigger corresponding to each XBAR trigger sources.

Note: This function provides a workaround implementation for ERR052412 by using blocking way to implement SW trigger.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.

```
uint32_t ADC_ETC_GetADCConversionValue(ADC_ETC_Type *base, uint32_t triggerGroup,
                                         uint32_t chainGroup)
```

Get ADC conversion result from external XBAR sources. For example, if triggerGroup is set to 0U and chainGroup is set to 1U, which means the API would return Trigger0 source's chain1 conversion result.

Parameters

- base – ADC_ETC peripheral base address.
- triggerGroup – Trigger group index. Available number is 0~7.
- chainGroup – Trigger chain group index. Available number is 0~7.

Returns

ADC conversion result value.

```
enum _adc_etc_status_flag_mask
```

ADC_ETC customized status flags mask.

Values:

enumerator kADC_ETC_Done0StatusFlagMask

enumerator kADC_ETC_Done1StatusFlagMask

enumerator kADC_ETC_Done2StatusFlagMask

enumerator kADC_ETC_Done3StatusFlagMask

enumerator kADC_ETC_ErrorStatusFlagMask

```
enum _adc_etc_external_trigger_source
```

External triggers sources.

Values:

enumerator kADC_ETC_Trg0TriggerSource

enumerator kADC_ETC_Trg1TriggerSource
enumerator kADC_ETC_Trg2TriggerSource
enumerator kADC_ETC_Trg3TriggerSource
enumerator kADC_ETC_Trg4TriggerSource
enumerator kADC_ETC_Trg5TriggerSource
enumerator kADC_ETC_Trg6TriggerSource
enumerator kADC_ETC_Trg7TriggerSource
enumerator kADC_ETC_TSC0TriggerSource
enumerator kADC_ETC_TSC1TriggerSource

enum _adc_etc_interrupt_enable
Interrupt enable/disable mask.

Values:

enumerator kADC_ETC_Done0InterruptEnable
enumerator kADC_ETC_Done1InterruptEnable
enumerator kADC_ETC_Done2InterruptEnable
enumerator kADC_ETC_Done3InterruptEnable

enum _adc_etc_dma_mode_selection
DMA mode selection.

Values:

enumerator kADC_ETC_TrigDMAWithLatchedSignal
enumerator kADC_ETC_TrigDMAWithPulsedSignal

typedef enum *_adc_etc_external_trigger_source* adc_etc_external_trigger_source_t
External triggers sources.

typedef enum *_adc_etc_interrupt_enable* adc_etc_interrupt_enable_t
Interrupt enable/disable mask.

typedef enum *_adc_etc_dma_mode_selection* adc_etc_dma_mode_selection_t
DMA mode selection.

typedef struct *_adc_etc_config* adc_etc_config_t
ADC_ETC configuration.

typedef struct *_adc_etc_trigger_chain_config* adc_etc_trigger_chain_config_t
ADC_ETC trigger chain configuration.

typedef struct *_adc_etc_trigger_config* adc_etc_trigger_config_t
ADC_ETC trigger configuration.

FSL_ADC_ETC_DRIVER_VERSION
ADC_ETC driver version.
Version 2.3.2.

ADC_ETC_DMA_CTRL_TRGn_REQ_MASK
The mask of status flags cleared by writing 1.


```

struct _adc_etc_config
    #include <fsl_adc_etc.h> ADC_ETC configuration.

struct _adc_etc_trigger_chain_config
    #include <fsl_adc_etc.h> ADC_ETC trigger chain configuration.

struct _adc_etc_trigger_config
    #include <fsl_adc_etc.h> ADC_ETC trigger configuration.

```

2.3 AIPSTZ: AHB to IP Bridge

```

void AIPSTZ_SetMasterPrivilegeLevel(AIPSTZ_Type *base, aipstz_master_t master, uint32_t
    privilegeConfig)

```

Configure the privilege level for master.

Parameters

- base – AIPSTZ peripheral base pointer
- master – Masters for AIPSTZ.
- privilegeConfig – Configuration is ORed from aipstz_master_privilege_level_t.

```

void AIPSTZ_SetPeripheralAccessControl(AIPSTZ_Type *base, aipstz_peripheral_t peripheral,
    uint32_t accessControl)

```

Configure the access for peripheral.

Parameters

- base – AIPSTZ peripheral base pointer
- peripheral – Peripheral for AIPSTZ.
- accessControl – Configuration is ORed from aipstz_peripheral_access_control_t.

FSL_AIPSTZ_DRIVER_VERSION

Version 2.0.1

```

enum _aipstz_master_privilege_level
    List of AIPSTZ privilege configuration.

```

Values:

```

enumerator kAIPSTZ_MasterBufferedWriteEnable
    Write accesses from this master are allowed to be buffered.

```

```

enumerator kAIPSTZ_MasterTrustedForReadEnable
    This master is trusted for read accesses.

```

```

enumerator kAIPSTZ_MasterTrustedForWriteEnable
    This master is trusted for write accesses.

```

```

enumerator kAIPSTZ_MasterForceUserModeEnable
    Accesses from this master are forced to user-mode.

```

```

enum _aipstz_master
    List of AIPSTZ masters. Organized by width for the 8-15 bits and shift for lower 8 bits.

```

Values:

```

enumerator kAIPSTZ_Master0

```

enumerator kAIPSTZ_Master1

enumerator kAIPSTZ_Master2

enumerator kAIPSTZ_Master3

enumerator kAIPSTZ_Master5

enum _aipstz_peripheral_access_control

List of AIPSTZ peripheral access control configuration.

Values:

enumerator kAIPSTZ_PeripheralAllowUntrustedMaster

enumerator kAIPSTZ_PeripheralWriteProtected

enumerator kAIPSTZ_PeripheralRequireSupervisor

enumerator kAIPSTZ_PeripheralAllowBufferedWrite

enum _aipstz_peripheral

List of AIPSTZ peripherals. Organized by register offset for higher 32 bits, width for the 8-15 bits and shift for lower 8 bits.

Values:

enumerator kAIPSTZ_Peripheral0

enumerator kAIPSTZ_Peripheral1

enumerator kAIPSTZ_Peripheral2

enumerator kAIPSTZ_Peripheral3

enumerator kAIPSTZ_Peripheral4

enumerator kAIPSTZ_Peripheral5

enumerator kAIPSTZ_Peripheral6

enumerator kAIPSTZ_Peripheral7

enumerator kAIPSTZ_Peripheral8

enumerator kAIPSTZ_Peripheral9

enumerator kAIPSTZ_Peripheral10

enumerator kAIPSTZ_Peripheral11

enumerator kAIPSTZ_Peripheral12

enumerator kAIPSTZ_Peripheral13

enumerator kAIPSTZ_Peripheral14

enumerator kAIPSTZ_Peripheral15

enumerator kAIPSTZ_Peripheral16

enumerator kAIPSTZ_Peripheral17

enumerator kAIPSTZ_Peripheral18

enumerator kAIPSTZ_Peripheral19

enumerator kAIPSTZ_Peripheral20

enumerator kAIPSTZ_Peripheral21

enumerator kAIPSTZ_Peripheral22

enumerator kAIPSTZ_Peripheral23

enumerator kAIPSTZ_Peripheral24

enumerator kAIPSTZ_Peripheral25

enumerator kAIPSTZ_Peripheral26

enumerator kAIPSTZ_Peripheral27

enumerator kAIPSTZ_Peripheral28

enumerator kAIPSTZ_Peripheral29

enumerator kAIPSTZ_Peripheral30

enumerator kAIPSTZ_Peripheral31

enumerator kAIPSTZ_Peripheral32

enumerator kAIPSTZ_Peripheral33

typedef enum *aipstz_master_privilege_level* aipstz_master_privilege_level_t

List of AIPSTZ privilege configuration.

typedef enum *aipstz_master* aipstz_master_t

List of AIPSTZ masters. Organized by width for the 8-15 bits and shift for lower 8 bits.

typedef enum *aipstz_peripheral_access_control* aipstz_peripheral_access_control_t

List of AIPSTZ peripheral access control configuration.

typedef enum *aipstz_peripheral* aipstz_peripheral_t

List of AIPSTZ peripherals. Organized by register offset for higher 32 bits, width for the 8-15 bits and shift for lower 8 bits.

2.4 AOI: Crossbar AND/OR/INVERT Driver

void AOI_Init(AOI_Type *base)

Initializes an AOI instance for operation.

This function un-gates the AOI clock.

Parameters

- base – AOI peripheral address.

void AOI_Deinit(AOI_Type *base)

Deinitializes an AOI instance for operation.

This function shutdowns AOI module.

Parameters

- base – AOI peripheral address.

```
void AOI_GetEventLogicConfig(AOI_Type *base, aoi_event_t event, aoi_event_config_t *config)
```

Gets the Boolean evaluation associated.

This function returns the Boolean evaluation associated.

Example:

```
aoi_event_config_t demoEventLogicStruct;

AOI_GetEventLogicConfig(AOI, kAOI_Event0, &demoEventLogicStruct);
```

Parameters

- base – AOI peripheral address.
- event – Index of the event which will be set of type `aoi_event_t`.
- config – Selected input configuration .

```
void AOI_SetEventLogicConfig(AOI_Type *base, aoi_event_t event, const aoi_event_config_t *eventConfig)
```

Configures an AOI event.

This function configures an AOI event according to the `aoiEventConfig` structure. This function configures all inputs (A, B, C, and D) of all product terms (0, 1, 2, and 3) of a desired event.

Example:

```
aoi_event_config_t demoEventLogicStruct;

demoEventLogicStruct.PT0AC = kAOI_InvInputSignal;
demoEventLogicStruct.PT0BC = kAOI_InputSignal;
demoEventLogicStruct.PT0CC = kAOI_LogicOne;
demoEventLogicStruct.PT0DC = kAOI_LogicOne;

demoEventLogicStruct.PT1AC = kAOI_LogicZero;
demoEventLogicStruct.PT1BC = kAOI_LogicOne;
demoEventLogicStruct.PT1CC = kAOI_LogicOne;
demoEventLogicStruct.PT1DC = kAOI_LogicOne;

demoEventLogicStruct.PT2AC = kAOI_LogicZero;
demoEventLogicStruct.PT2BC = kAOI_LogicOne;
demoEventLogicStruct.PT2CC = kAOI_LogicOne;
demoEventLogicStruct.PT2DC = kAOI_LogicOne;

demoEventLogicStruct.PT3AC = kAOI_LogicZero;
demoEventLogicStruct.PT3BC = kAOI_LogicOne;
demoEventLogicStruct.PT3CC = kAOI_LogicOne;
demoEventLogicStruct.PT3DC = kAOI_LogicOne;

AOI_SetEventLogicConfig(AOI, kAOI_Event0, demoEventLogicStruct);
```

Parameters

- base – AOI peripheral address.
- event – Event which will be configured of type `aoi_event_t`.
- eventConfig – Pointer to type `aoi_event_config_t` structure. The user is responsible for filling out the members of this structure and passing the pointer to this function.

FSL_AOI_DRIVER_VERSION

Version 2.0.2.

enum `_aoi_input_config`

AOI input configurations.

The selection item represents the Boolean evaluations.

Values:

enumerator `kAOI_LogicZero`

Forces the input to logical zero.

enumerator `kAOI_InputSignal`

Passes the input signal.

enumerator `kAOI_InvInputSignal`

Inverts the input signal.

enumerator `kAOI_LogicOne`

Forces the input to logical one.

enum `_aoi_event`

AOI event indexes, where an event is the collection of the four product terms (0, 1, 2, and 3) and the four signal inputs (A, B, C, and D).

Values:

enumerator `kAOI_Event0`

Event 0 index

enumerator `kAOI_Event1`

Event 1 index

enumerator `kAOI_Event2`

Event 2 index

enumerator `kAOI_Event3`

Event 3 index

typedef enum `_aoi_input_config` `aoi_input_config_t`

AOI input configurations.

The selection item represents the Boolean evaluations.

typedef enum `_aoi_event` `aoi_event_t`

AOI event indexes, where an event is the collection of the four product terms (0, 1, 2, and 3) and the four signal inputs (A, B, C, and D).

typedef struct `_aoi_event_config` `aoi_event_config_t`

AOI event configuration structure.

Defines structure `_aoi_event_config` and use the `AOI_SetEventLogicConfig()` function to make whole event configuration.

AOI

AOI peripheral address

struct `_aoi_event_config`

`#include <fsl_aoi.h>` AOI event configuration structure.

Defines structure `_aoi_event_config` and use the `AOI_SetEventLogicConfig()` function to make whole event configuration.

Public Members

aoi_input_config_t PT0AC
Product term 0 input A

aoi_input_config_t PT0BC
Product term 0 input B

aoi_input_config_t PT0CC
Product term 0 input C

aoi_input_config_t PT0DC
Product term 0 input D

aoi_input_config_t PT1AC
Product term 1 input A

aoi_input_config_t PT1BC
Product term 1 input B

aoi_input_config_t PT1CC
Product term 1 input C

aoi_input_config_t PT1DC
Product term 1 input D

aoi_input_config_t PT2AC
Product term 2 input A

aoi_input_config_t PT2BC
Product term 2 input B

aoi_input_config_t PT2CC
Product term 2 input C

aoi_input_config_t PT2DC
Product term 2 input D

aoi_input_config_t PT3AC
Product term 3 input A

aoi_input_config_t PT3BC
Product term 3 input B

aoi_input_config_t PT3CC
Product term 3 input C

aoi_input_config_t PT3DC
Product term 3 input D

2.5 CACHE: ARMV7-M7 CACHE Memory Controller

static inline void L1CACHE__EnableICache(void)
Enables cortex-m7 L1 instruction cache.

static inline void L1CACHE__DisableICache(void)
Disables cortex-m7 L1 instruction cache.

static inline void L1CACHE__InvalidateICache(void)
Invalidate cortex-m7 L1 instruction cache.

`void L1CACHE_InvalidateICacheByRange(uint32_t address, uint32_t size_byte)`

Invalidate cortex-m7 L1 instruction cache by range.

Note: The start address and size_byte should be 32-byte(FSL_FEATURE_L1ICACHE_LINESIZE_BYTE) aligned. The startAddr here will be forced to align to L1 I-cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

`static inline void L1CACHE_EnableDCache(void)`

Enables cortex-m7 L1 data cache.

`static inline void L1CACHE_DisableDCache(void)`

Disables cortex-m7 L1 data cache.

`static inline void L1CACHE_InvalidateDCache(void)`

Invalidates cortex-m7 L1 data cache.

`static inline void L1CACHE_CleanDCache(void)`

Cleans cortex-m7 L1 data cache.

`static inline void L1CACHE_CleanInvalidateDCache(void)`

Cleans and Invalidates cortex-m7 L1 data cache.

`static inline void L1CACHE_InvalidateDCacheByRange(uint32_t address, uint32_t size_byte)`

Invalidates cortex-m7 L1 data cache by range.

Note: The start address and size_byte should be 32-byte(FSL_FEATURE_L1DCACHE_LINESIZE_BYTE) aligned. The startAddr here will be forced to align to L1 D-cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

`static inline void L1CACHE_CleanDCacheByRange(uint32_t address, uint32_t size_byte)`

Cleans cortex-m7 L1 data cache by range.

Note: The start address and size_byte should be 32-byte(FSL_FEATURE_L1DCACHE_LINESIZE_BYTE) aligned. The startAddr here will be forced to align to L1 D-cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

- size_byte – The memory size.

```
static inline void L1CACHE_CleanInvalidateDCacheByRange(uint32_t address, uint32_t
                                                         size_byte)
```

Cleans and Invalidates cortex-m7 L1 data cache by range.

Note: The start address and size_byte should be 32-byte(FSL_FEATURE_L1DCACHE_LINESIZE_BYTE) aligned. The startAddr here will be forced to align to L1 D-cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

```
void ICACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)
```

Invalidates all instruction caches by range.

Both cortex-m7 L1 cache line and L2 PL310 cache line length is 32-byte.

Note: address and size should be aligned to cache line size 32-Byte due to the cache operation unit is one cache line. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

```
void DCACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)
```

Invalidates all data caches by range.

Both cortex-m7 L1 cache line and L2 PL310 cache line length is 32-byte.

Note: address and size should be aligned to cache line size 32-Byte due to the cache operation unit is one cache line. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

```
void DCACHE_CleanByRange(uint32_t address, uint32_t size_byte)
```

Cleans all data caches by range.

Both cortex-m7 L1 cache line and L2 PL310 cache line length is 32-byte.

Note: address and size should be aligned to cache line size 32-Byte due to the cache operation unit is one cache line. The startAddr here will be forced to align to the cache line size

if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

void DCACHE_CleanInvalidateByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates all data caches by range.

Both cortex-m7 L1 cache line and L2 PL310 cache line length is 32-byte.

Note: address and size should be aligned to cache line size 32-Byte due to the cache operation unit is one cache line. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

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

FSL_CACHE_DRIVER_VERSION

cache driver version 2.0.4.

2.6 Clock Driver

enum _clock_name

Clock name used to get clock frequency.

Values:

enumerator kCLOCK_CpuClk

CPU clock

enumerator kCLOCK_CoreClk

CORE clock

enumerator kCLOCK_IpgClk

IPG clock

enumerator kCLOCK_PerClk

PER clock

enumerator kCLOCK_OscClk

OSC clock selected by PMU_LOWPWR_CTRL[OSC_SEL].

enumerator kCLOCK_RtcClk

RTC clock. (RTCCLK)

enumerator kCLOCK_Usb1PllClk

USB1PLLCLK.

enumerator kCLOCK_Usb1PllPfd0Clk

USB1PLLPDF0CLK.

enumerator kCLOCK__Usb1PlIPfd1Clk
USB1PLLPDF1CLK.

enumerator kCLOCK__Usb1PlIPfd2Clk
USB1PLLPDF2CLK.

enumerator kCLOCK__Usb1PlIPfd3Clk
USB1PLLPDF3CLK.

enumerator kCLOCK__Usb1SwClk
USB1PLLSWCLK

enumerator kCLOCK__Usb1Sw60MClk
USB1PLLSw60MCLK

enumerator kCLOCK__Usb1Sw80MClk
USB1PLLSw80MCLK

enumerator kCLOCK__SysPlIClk
SYSPLLCLK.

enumerator kCLOCK__SysPlIPfd0Clk
SYSPLLPDF0CLK.

enumerator kCLOCK__SysPlIPfd1Clk
SYSPLLPDF1CLK.

enumerator kCLOCK__SysPlIPfd2Clk
SYSPLLPDF2CLK.

enumerator kCLOCK__SysPlIPfd3Clk
SYSPLLPDF3CLK.

enumerator kCLOCK__EnetPlI500MClk
Enet PLLCLK ref_enetpll500M.

enumerator kCLOCK__AudioPlIClk
Audio PLLCLK.

enumerator kCLOCK__PeriphClk2
Periph CLK2 selection.

enumerator kCLOCK__FlexspiSel
Flexspi selection.

enumerator kCLOCK__NoneName
None Clock Name.

enum __clock_ip_name
CCM CCGR gate control for each module independently.

Values:

enumerator kCLOCK__IpInvalid

enumerator kCLOCK__Aips__tz1
CCGR0, CG0

enumerator kCLOCK__Aips__tz2
CCGR0, CG1

enumerator kCLOCK__Mqs
CCGR0, CG2

enumerator kCLOCK_FlexSpiExsc
CCGR0, CG3

enumerator kCLOCK_Sim_m_clk_r
CCGR0, CG4

enumerator kCLOCK_Dcp
CCGR0, CG5

enumerator kCLOCK_Lpuart3
CCGR0, CG6

enumerator kCLOCK_Trace
CCGR0, CG11

enumerator kCLOCK_Gpt2
CCGR0, CG12

enumerator kCLOCK_Gpt2S
CCGR0, CG13

enumerator kCLOCK_Lpuart2
CCGR0, CG14

enumerator kCLOCK_Gpio2
CCGR0, CG15

enumerator kCLOCK_Lpspi1
CCGR1, CG0

enumerator kCLOCK_Lpspi2
CCGR1, CG1

enumerator kCLOCK_Pit
CCGR1, CG6

enumerator kCLOCK_Adc1
CCGR1, CG8

enumerator kCLOCK_Gpt1
CCGR1, CG10

enumerator kCLOCK_Gpt1S
CCGR1, CG11

enumerator kCLOCK_Lpuart4
CCGR1, CG12

enumerator kCLOCK_Gpio1
CCGR1, CG13

enumerator kCLOCK_Csu
CCGR1, CG14

enumerator kCLOCK_Gpio5
CCGR1, CG15

enumerator kCLOCK_OcramExsc
CCGR2, CG0

enumerator kCLOCK_IomuxcSnvs
CCGR2, CG2

enumerator kCLOCK_Lpi2c1
CCGR2, CG3

enumerator kCLOCK_Lpi2c2
CCGR2, CG4

enumerator kCLOCK_Ocotp
CCGR2, CG6

enumerator kCLOCK_Xbar1
CCGR2, CG11

enumerator kCLOCK_Aoi
CCGR3, CG4

enumerator kCLOCK_Ewm0
CCGR3, CG7

enumerator kCLOCK_Wdog1
CCGR3, CG8

enumerator kCLOCK_FlexRam
CCGR3, CG9

enumerator kCLOCK_IomuxcSnvsGpr
CCGR3, CG15

enumerator kCLOCK_Sim_m7_clk_r
CCGR4, CG0

enumerator kCLOCK_Iomuxc
CCGR4, CG1

enumerator kCLOCK_IomuxcGpr
CCGR4, CG2

enumerator kCLOCK_SimM7
CCGR4, CG4

enumerator kCLOCK_SimM
CCGR4, CG6

enumerator kCLOCK_SimEms
CCGR4, CG7

enumerator kCLOCK_Pwm1
CCGR4, CG8

enumerator kCLOCK_Dma_ps
CCGR4, CG15,

enumerator kCLOCK_Rom
CCGR5, CG0

enumerator kCLOCK_Flexio1
CCGR5, CG1

enumerator kCLOCK_Wdog3
CCGR5, CG2

enumerator kCLOCK_Dma
CCGR5, CG3

enumerator kCLOCK_Kpp
CCGR5, CG4

enumerator kCLOCK_Wdog2
CCGR5, CG5

enumerator kCLOCK_Spdif
CCGR5, CG7

enumerator kCLOCK_Sai1
CCGR5, CG9

enumerator kCLOCK_Sai3
CCGR5, CG11

enumerator kCLOCK_Lpuart1
CCGR5, CG12

enumerator kCLOCK_SnvsHp
CCGR5, CG14

enumerator kCLOCK_SnvsLp
CCGR5, CG15

enumerator kCLOCK_UsbOh3
CCGR6, CG0

enumerator kCLOCK_Dcdc
CCGR6, CG3

enumerator kCLOCK_FlexSpi
CCGR6, CG5

enumerator kCLOCK_Trng
CCGR6, CG6

enumerator kCLOCK_SimPer
CCGR6, CG10

enumerator kCLOCK_Anadig
CCGR6, CG11

enum _clock_osc
OSC 24M source select.

Values:

enumerator kCLOCK_RcOsc
On chip OSC.

enumerator kCLOCK_XtalOsc
24M Xtal OSC

enum _clock_gate_value
Clock gate value.

Values:

enumerator kCLOCK_ClockNotNeeded
Clock is off during all modes.

enumerator kCLOCK_ClockNeededRun
Clock is on in run mode, but off in WAIT and STOP modes

enumerator kCLOCK_ClockNeededRunWait

Clock is on during all modes, except STOP mode

enum _clock_mode_t

System clock mode.

Values:

enumerator kCLOCK_ModeRun

Remain in run mode.

enumerator kCLOCK_ModeWait

Transfer to wait mode.

enumerator kCLOCK_ModeStop

Transfer to stop mode.

enum _clock_mux

MUX control names for clock mux setting.

These constants define the mux control names for clock mux setting.

- 0:7: REG offset to CCM_BASE in bytes.
- 8:15: Root clock setting bit field shift.
- 16:31: Root clock setting bit field width.

Values:

enumerator kCLOCK_Pll3SwMux

pll3_sw_clk mux name

enumerator kCLOCK_PeriphMux

periph mux name

enumerator kCLOCK_PrePeriphMux

pre-periph mux name

enumerator kCLOCK_TraceMux

trace mux name

enumerator kCLOCK_PeriphClk2Mux

periph clock2 mux name

enumerator kCLOCK_LpspiMux

lpspi mux name

enumerator kCLOCK_FlexspiMux

flexspi mux name

enumerator kCLOCK_FlexspiSrcMux

flexspi SRC mux name

enumerator kCLOCK_Sai3Mux

sai3 mux name

enumerator kCLOCK_Sai1Mux

sai1 mux name

enumerator kCLOCK_PerclkMux

perclk mux name

enumerator kCLOCK_Flexio1Mux

flexio1 mux name

enumerator kCLOCK__UartMux

uart mux name

enumerator kCLOCK__SpdifMux

spdif mux name

enumerator kCLOCK__Lpi2cMux

lpi2c mux name

enum __clock_div

DIV control names for clock div setting.

These constants define div control names for clock div setting.

- 0:7: REG offset to CCM_BASE in bytes.
- 8:15: Root clock setting bit field shift.
- 16:31: Root clock setting bit field width.

Values:

enumerator kCLOCK__AhbDiv

ahb div name

enumerator kCLOCK__IpgDiv

ipg div name

enumerator kCLOCK__LpspiDiv

lpspi div name

enumerator kCLOCK__FlexspiDiv

flexspi div name

enumerator kCLOCK__PerclkDiv

perclk div name

enumerator kCLOCK__AdcDiv

adc name

enumerator kCLOCK__TraceDiv

trace div name

enumerator kCLOCK__UartDiv

uart div name

enumerator kCLOCK__Flexio1Div

flexio1 pre div name

enumerator kCLOCK__Sai3PreDiv

sai3 pre div name

enumerator kCLOCK__Sai3Div

sai3 div name

enumerator kCLOCK__Flexio1PreDiv

flexio1 pre div name

enumerator kCLOCK__Sai1PreDiv

sai1 pre div name

enumerator kCLOCK__Sai1Div

sai1 div name

enumerator kCLOCK_Spdif0PreDiv
 spdif pre div name

enumerator kCLOCK_Spdif0Div
 spdif div name

enumerator kCLOCK_Lpi2cDiv
 lpi2c div name

enumerator kCLOCK_NonePreDiv
 None Pre div.

enum _clock_div_value
 Clock divider value.

Values:

enumerator kCLOCK_AhbDivBy1
 Ahb clock divider set to divided by 1.

enumerator kCLOCK_AhbDivBy2
 Ahb clock divider set to divided by 2.

enumerator kCLOCK_AhbDivBy3
 Ahb clock divider set to divided by 3.

enumerator kCLOCK_AhbDivBy4
 Ahb clock divider set to divided by 4.

enumerator kCLOCK_AhbDivBy5
 Ahb clock divider set to divided by 5.

enumerator kCLOCK_AhbDivBy6
 Ahb clock divider set to divided by 6.

enumerator kCLOCK_AhbDivBy7
 Ahb clock divider set to divided by 7.

enumerator kCLOCK_AhbDivBy8
 Ahb clock divider set to divided by 8.

enumerator kCLOCK_IpgDivBy1
 ipg clock divider set to divided by 1.

enumerator kCLOCK_IpgDivBy2
 ipg clock divider set to divided by 2.

enumerator kCLOCK_IpgDivBy3
 ipg clock divider set to divided by 3.

enumerator kCLOCK_IpgDivBy4
 ipg clock divider set to divided by 4.

enumerator kCLOCK_LpspiDivBy1
 lpspi clock divider set to divided by 1.

enumerator kCLOCK_LpspiDivBy2
 lpspi clock divider set to divided by 2.

enumerator kCLOCK_LpspiDivBy3
 lpspi clock divider set to divided by 3.

enumerator kCLOCK_LpspiDivBy4
lpspi clock divider set to divided by 4.

enumerator kCLOCK_LpspiDivBy5
lpspi clock divider set to divided by 5.

enumerator kCLOCK_LpspiDivBy6
lpspi clock divider set to divided by 6.

enumerator kCLOCK_LpspiDivBy7
lpspi clock divider set to divided by 7.

enumerator kCLOCK_LpspiDivBy8
lpspi clock divider set to divided by 8.

enumerator kCLOCK_LpspiDivBy9
lpspi clock divider set to divided by 9.

enumerator kCLOCK_LpspiDivBy10
lpspi clock divider set to divided by 10.

enumerator kCLOCK_LpspiDivBy11
lpspi clock divider set to divided by 11.

enumerator kCLOCK_LpspiDivBy12
lpspi clock divider set to divided by 12.

enumerator kCLOCK_LpspiDivBy13
lpspi clock divider set to divided by 13.

enumerator kCLOCK_LpspiDivBy14
lpspi clock divider set to divided by 14.

enumerator kCLOCK_LpspiDivBy15
lpspi clock divider set to divided by 15.

enumerator kCLOCK_LpspiDivBy16
lpspi clock divider set to divided by 16.

enumerator kCLOCK_FlexspiDivBy1
flexspi clock divider set to divided by 1.

enumerator kCLOCK_FlexspiDivBy2
flexspi clock divider set to divided by 2.

enumerator kCLOCK_FlexspiDivBy3
flexspi clock divider set to divided by 3.

enumerator kCLOCK_FlexspiDivBy4
flexspi clock divider set to divided by 4.

enumerator kCLOCK_FlexspiDivBy5
flexspi clock divider set to divided by 5.

enumerator kCLOCK_FlexspiDivBy6
flexspi clock divider set to divided by 6.

enumerator kCLOCK_FlexspiDivBy7
flexspi clock divider set to divided by 7.

enumerator kCLOCK_FlexspiDivBy8
flexspi clock divider set to divided by 8.

enumerator kCLOCK_AdcDivBy8
adc clock divider set to divided by 8.

enumerator kCLOCK_AdcDivBy12
adc clock divider set to divided by 12.

enumerator kCLOCK_AdcDivBy16
adc clock divider set to divided by 16.

enumerator kCLOCK_TraceDivBy1
trace clock divider set to divided by 1.

enumerator kCLOCK_TraceDivBy2
trace clock divider set to divided by 2.

enumerator kCLOCK_TraceDivBy3
trace clock divider set to divided by 3.

enumerator kCLOCK_TraceDivBy4
trace clock divider set to divided by 4.

enumerator kCLOCK_TraceDivBy5
trace clock divider set to divided by 5.

enumerator kCLOCK_TraceDivBy6
trace clock divider set to divided by 6.

enumerator kCLOCK_TraceDivBy7
trace clock divider set to divided by 7.

enumerator kCLOCK_TraceDivBy8
trace clock divider set to divided by 8.

enumerator kCLOCK_TraceDivBy9
trace clock divider set to divided by 9.

enumerator kCLOCK_TraceDivBy10
trace clock divider set to divided by 10.

enumerator kCLOCK_TraceDivBy11
trace clock divider set to divided by 11.

enumerator kCLOCK_TraceDivBy12
trace clock divider set to divided by 12.

enumerator kCLOCK_TraceDivBy13
trace clock divider set to divided by 13.

enumerator kCLOCK_TraceDivBy14
trace clock divider set to divided by 14.

enumerator kCLOCK_TraceDivBy15
trace clock divider set to divided by 15.

enumerator kCLOCK_TraceDivBy16
trace clock divider set to divided by 16.

enumerator kCLOCK_Flexio1DivBy1
flexio1 clock divider set to divided by 1.

enumerator kCLOCK_Flexio1DivBy2
flexio1 clock divider set to divided by 2.

enumerator kCLOCK_Flexio1DivBy3
flexio1 clock divider set to divided by 3.

enumerator kCLOCK_Flexio1DivBy4
flexio1 clock divider set to divided by 4.

enumerator kCLOCK_Flexio1DivBy5
flexio1 clock divider set to divided by 5.

enumerator kCLOCK_Flexio1DivBy6
flexio1 clock divider set to divided by 6.

enumerator kCLOCK_Flexio1DivBy7
flexio1 clock divider set to divided by 7.

enumerator kCLOCK_Flexio1DivBy8
flexio1 clock divider set to divided by 8.

enumerator kCLOCK_Flexio1DivBy9
flexio1 clock divider set to divided by 9.

enumerator kCLOCK_Flexio1DivBy10
flexio1 clock divider set to divided by 10.

enumerator kCLOCK_Flexio1DivBy11
flexio1 clock divider set to divided by 11.

enumerator kCLOCK_Flexio1DivBy12
flexio1 clock divider set to divided by 12.

enumerator kCLOCK_Flexio1DivBy13
flexio1 clock divider set to divided by 13.

enumerator kCLOCK_Flexio1DivBy14
flexio1 clock divider set to divided by 14.

enumerator kCLOCK_Flexio1DivBy15
flexio1 clock divider set to divided by 15.

enumerator kCLOCK_Flexio1DivBy16
flexio1 clock divider set to divided by 16.

enumerator kCLOCK_Sai3PreDivBy1
sai3 pre clock divider set to divided by 1.

enumerator kCLOCK_Sai3PreDivBy2
sai3 pre clock divider set to divided by 2.

enumerator kCLOCK_Sai3PreDivBy3
sai3 pre clock divider set to divided by 3.

enumerator kCLOCK_Sai3PreDivBy4
sai3 pre clock divider set to divided by 4.

enumerator kCLOCK_Sai3PreDivBy5
sai3 pre clock divider set to divided by 5.

enumerator kCLOCK_Sai3PreDivBy6
sai3 pre clock divider set to divided by 6.

enumerator kCLOCK_Sai3PreDivBy7
sai3 pre clock divider set to divided by 7.

enumerator kCLOCK_Sai3PreDivBy8
sai3 pre clock divider set to divided by 8.

enumerator kCLOCK_Flexio1PreDivBy1
flexio1 pre clock divider set to divided by 1.

enumerator kCLOCK_Flexio1PreDivBy2
flexio1 pre clock divider set to divided by 2.

enumerator kCLOCK_Flexio1PreDivBy3
flexio1 pre clock divider set to divided by 3.

enumerator kCLOCK_Flexio1PreDivBy4
flexio1 pre clock divider set to divided by 4.

enumerator kCLOCK_Flexio1PreDivBy5
flexio1 pre clock divider set to divided by 5.

enumerator kCLOCK_Flexio1PreDivBy6
flexio1 pre clock divider set to divided by 6.

enumerator kCLOCK_Flexio1PreDivBy7
flexio1 pre clock divider set to divided by 7.

enumerator kCLOCK_Flexio1PreDivBy8
flexio1 pre clock divider set to divided by 8.

enumerator kCLOCK_Sai1PreDivBy1
sai1 pre clock divider set to divided by 1.

enumerator kCLOCK_Sai1PreDivBy2
sai1 pre clock divider set to divided by 2.

enumerator kCLOCK_Sai1PreDivBy3
sai1 pre clock divider set to divided by 3.

enumerator kCLOCK_Sai1PreDivBy4
sai1 pre clock divider set to divided by 4.

enumerator kCLOCK_Sai1PreDivBy5
sai1 pre clock divider set to divided by 5.

enumerator kCLOCK_Sai1PreDivBy6
sai1 pre clock divider set to divided by 6.

enumerator kCLOCK_Sai1PreDivBy7
sai1 pre clock divider set to divided by 7.

enumerator kCLOCK_Sai1PreDivBy8
sai1 pre clock divider set to divided by 8.

enumerator kCLOCK_Spdif0PreDivBy1
spdif pre clock divider set to divided by 1.

enumerator kCLOCK_Spdif0PreDivBy2
spdif pre clock divider set to divided by 2.

enumerator kCLOCK_Spdif0PreDivBy3
spdif pre clock divider set to divided by 3.

enumerator kCLOCK_Spdif0PreDivBy4
spdif pre clock divider set to divided by 4.

enumerator kCLOCK_Spdif0PreDivBy5
 spdif pre clock divider set to divided by 5.

enumerator kCLOCK_Spdif0PreDivBy6
 spdif pre clock divider set to divided by 6.

enumerator kCLOCK_Spdif0PreDivBy7
 spdif pre clock divider set to divided by 7.

enumerator kCLOCK_Spdif0PreDivBy8
 spdif pre clock divider set to divided by 8.

enumerator kCLOCK_Spdif0DivBy1
 spdif clock divider set to divided by 1.

enumerator kCLOCK_Spdif0DivBy2
 spdif clock divider set to divided by 2.

enumerator kCLOCK_Spdif0DivBy3
 spdif clock divider set to divided by 3.

enumerator kCLOCK_Spdif0DivBy4
 spdif clock divider set to divided by 4.

enumerator kCLOCK_Spdif0DivBy5
 spdif clock divider set to divided by 5.

enumerator kCLOCK_Spdif0DivBy6
 spdif clock divider set to divided by 6.

enumerator kCLOCK_Spdif0DivBy7
 spdif clock divider set to divided by 7.

enumerator kCLOCK_Spdif0DivBy8
 spdif clock divider set to divided by 8.

enumerator kCLOCK_MiscDivBy1
 Misc divider like LPI2C set to divided by 1.

enumerator kCLOCK_MiscDivBy2
 Misc divider like LPI2C set to divided by 2.

enumerator kCLOCK_MiscDivBy3
 Misc divider like LPI2C set to divided by 3.

enumerator kCLOCK_MiscDivBy4
 Misc divider like LPI2C set to divided by 4.

enumerator kCLOCK_MiscDivBy5
 Misc divider like LPI2C set to divided by 5.

enumerator kCLOCK_MiscDivBy6
 Misc divider like LPI2C set to divided by 6.

enumerator kCLOCK_MiscDivBy7
 Misc divider like LPI2C set to divided by 7.

enumerator kCLOCK_MiscDivBy8
 Misc divider like LPI2C set to divided by 8.

enumerator kCLOCK_MiscDivBy9
 Misc divider like LPI2C set to divided by 9.

enumerator kCLOCK_MiscDivBy10
Misc divider like LPI2C set to divided by 10.

enumerator kCLOCK_MiscDivBy11
Misc divider like LPI2C set to divided by 11.

enumerator kCLOCK_MiscDivBy12
Misc divider like LPI2C set to divided by 12.

enumerator kCLOCK_MiscDivBy13
Misc divider like LPI2C set to divided by 13.

enumerator kCLOCK_MiscDivBy14
Misc divider like LPI2C set to divided by 14.

enumerator kCLOCK_MiscDivBy15
Misc divider like LPI2C set to divided by 15.

enumerator kCLOCK_MiscDivBy16
Misc divider like LPI2C set to divided by 16.

enumerator kCLOCK_MiscDivBy17
Misc divider like LPI2C set to divided by 17.

enumerator kCLOCK_MiscDivBy18
Misc divider like LPI2C set to divided by 18.

enumerator kCLOCK_MiscDivBy19
Misc divider like LPI2C set to divided by 19.

enumerator kCLOCK_MiscDivBy20
Misc divider like LPI2C set to divided by 20.

enumerator kCLOCK_MiscDivBy21
Misc divider like LPI2C set to divided by 21.

enumerator kCLOCK_MiscDivBy22
Misc divider like LPI2C set to divided by 22.

enumerator kCLOCK_MiscDivBy23
Misc divider like LPI2C set to divided by 23.

enumerator kCLOCK_MiscDivBy24
Misc divider like LPI2C set to divided by 24.

enumerator kCLOCK_MiscDivBy25
Misc divider like LPI2C set to divided by 25.

enumerator kCLOCK_MiscDivBy26
Misc divider like LPI2C set to divided by 26.

enumerator kCLOCK_MiscDivBy27
Misc divider like LPI2C set to divided by 27.

enumerator kCLOCK_MiscDivBy28
Misc divider like LPI2C set to divided by 28.

enumerator kCLOCK_MiscDivBy29
Misc divider like LPI2C set to divided by 29.

enumerator kCLOCK_MiscDivBy30
Misc divider like LPI2C set to divided by 30.

enumerator kCLOCK_MiscDivBy31
Misc divider like LPI2C set to divided by 31.

enumerator kCLOCK_MiscDivBy32
Misc divider like LPI2C set to divided by 32.

enumerator kCLOCK_MiscDivBy33
Misc divider like LPI2C set to divided by 33.

enumerator kCLOCK_MiscDivBy34
Misc divider like LPI2C set to divided by 34.

enumerator kCLOCK_MiscDivBy35
Misc divider like LPI2C set to divided by 35.

enumerator kCLOCK_MiscDivBy36
Misc divider like LPI2C set to divided by 36.

enumerator kCLOCK_MiscDivBy37
Misc divider like LPI2C set to divided by 37.

enumerator kCLOCK_MiscDivBy38
Misc divider like LPI2C set to divided by 38.

enumerator kCLOCK_MiscDivBy39
Misc divider like LPI2C set to divided by 39.

enumerator kCLOCK_MiscDivBy40
Misc divider like LPI2C set to divided by 40.

enumerator kCLOCK_MiscDivBy41
Misc divider like LPI2C set to divided by 41.

enumerator kCLOCK_MiscDivBy42
Misc divider like LPI2C set to divided by 42.

enumerator kCLOCK_MiscDivBy43
Misc divider like LPI2C set to divided by 43.

enumerator kCLOCK_MiscDivBy44
Misc divider like LPI2C set to divided by 44.

enumerator kCLOCK_MiscDivBy45
Misc divider like LPI2C set to divided by 45.

enumerator kCLOCK_MiscDivBy46
Misc divider like LPI2C set to divided by 46.

enumerator kCLOCK_MiscDivBy47
Misc divider like LPI2C set to divided by 47.

enumerator kCLOCK_MiscDivBy48
Misc divider like LPI2C set to divided by 48.

enumerator kCLOCK_MiscDivBy49
Misc divider like LPI2C set to divided by 49.

enumerator kCLOCK_MiscDivBy50
Misc divider like LPI2C set to divided by 50.

enumerator kCLOCK_MiscDivBy51
Misc divider like LPI2C set to divided by 51.

enumerator kCLOCK_MiscDivBy52

Misc divider like LPI2C set to divided by 52.

enumerator kCLOCK_MiscDivBy53

Misc divider like LPI2C set to divided by 53.

enumerator kCLOCK_MiscDivBy54

Misc divider like LPI2C set to divided by 54.

enumerator kCLOCK_MiscDivBy55

Misc divider like LPI2C set to divided by 55.

enumerator kCLOCK_MiscDivBy56

Misc divider like LPI2C set to divided by 56.

enumerator kCLOCK_MiscDivBy57

Misc divider like LPI2C set to divided by 57.

enumerator kCLOCK_MiscDivBy58

Misc divider like LPI2C set to divided by 58.

enumerator kCLOCK_MiscDivBy59

Misc divider like LPI2C set to divided by 59.

enumerator kCLOCK_MiscDivBy60

Misc divider like LPI2C set to divided by 60.

enumerator kCLOCK_MiscDivBy61

Misc divider like LPI2C set to divided by 61.

enumerator kCLOCK_MiscDivBy62

Misc divider like LPI2C set to divided by 62.

enumerator kCLOCK_MiscDivBy63

Misc divider like LPI2C set to divided by 63.

enumerator kCLOCK_MiscDivBy64

Misc divider like LPI2C set to divided by 64.

enum _clock_usb_src

USB clock source definition.

Values:

enumerator kCLOCK_Usb480M

Use 480M.

enumerator kCLOCK_UsbSrcUnused

Used when the function does not care the clock source.

enum _clock_usb_phy_src

Source of the USB HS PHY.

Values:

enumerator kCLOCK_Usbphy480M

Use 480M.

enum _clock_pll_clk_src

PLL clock source, bypass cloco source also.

Values:

enumerator kCLOCK_PllClkSrc24M

PLL clock source 24M

enumerator kCLOCK_PllSrcClkPN

PLL clock source CLK1_P and CLK1_N

enum __clock_pll

PLL name.

Values:

enumerator kCLOCK_PllSys

PLL SYS

enumerator kCLOCK_PllUsb1

PLL USB1

enumerator kCLOCK_PllAudio

PLL Audio

enumerator kCLOCK_PllEnet500M

PLL ENET

enum __clock_pfd

PLL PFD name.

Values:

enumerator kCLOCK_Pfd0

PLL PFD0

enumerator kCLOCK_Pfd1

PLL PFD1

enumerator kCLOCK_Pfd2

PLL PFD2

enumerator kCLOCK_Pfd3

PLL PFD3

enum __clock_output1_selection

The enumerator of clock output1's clock source, such as USB1 PLL, SYS PLL and so on.

Values:

enumerator kCLOCK_OutputPllUsb1Sw

Selects USB1 PLL SW clock(Divided by 2) output.

enumerator kCLOCK_OutputPllSys

Selects SYS PLL clock(Divided by 2) output.

enumerator kCLOCK_OutputPllENET

Selects ENET PLL clock(Divided by 2) output.

enumerator kCLOCK_OutputCoreClk

Selects Core clock root output.

enumerator kCLOCK_OutputIpgClk

Selects IPG clock root output.

enumerator kCLOCK_OutputPerClk

Selects PERCLK clock root output.

enumerator kCLOCK__OutputPll4MainClk
Selects PLL4 main clock output.

enumerator kCLOCK__DisableClockOutput1
Disables CLK01.

enum __clock_output2_selection

The enumerator of clock output2's clock source, such as USDHC1 clock root, LPI2C clock root and so on.

Values:

enumerator kCLOCK__OutputLpi2cClk
Selects LPI2C clock root output.

enumerator kCLOCK__OutputOscClk
Selects OSC output.

enumerator kCLOCK__OutputLpspiClk
Selects LPSPi clock root output.

enumerator kCLOCK__OutputSai1Clk
Selects SAI1 clock root output.

enumerator kCLOCK__OutputSai3Clk
Selects SAI3 clock root output.

enumerator kCLOCK__OutputTraceClk
Selects Trace clock root output.

enumerator kCLOCK__OutputFlexspiClk
Selects FLEXSPi clock root output.

enumerator kCLOCK__OutputUartClk
Selects UART clock root output.

enumerator kCLOCK__OutputSpdif0Clk
Selects SPDIF0 clock root output.

enumerator kCLOCK__DisableClockOutput2
Disables CLK02.

enum __clock_output_divider

The enumerator of clock output's divider.

Values:

enumerator kCLOCK__DivideBy1
Output clock divided by 1.

enumerator kCLOCK__DivideBy2
Output clock divided by 2.

enumerator kCLOCK__DivideBy3
Output clock divided by 3.

enumerator kCLOCK__DivideBy4
Output clock divided by 4.

enumerator kCLOCK__DivideBy5
Output clock divided by 5.

enumerator kCLOCK__DivideBy6
Output clock divided by 6.

enumerator kCLOCK_DivideBy7

Output clock divided by 7.

enumerator kCLOCK_DivideBy8

Output clock divided by 8.

enum _clock_root

The enumerator of clock root.

Values:

enumerator kCLOCK_FlexspiClkRoot

FLEXSPI clock root.

enumerator kCLOCK_LpspiClkRoot

LPSPi clock root.

enumerator kCLOCK_TraceClkRoot

Trace clock root.

enumerator kCLOCK_Sai1ClkRoot

SAI1 clock root.

enumerator kCLOCK_Sai3ClkRoot

SAI3 clock root.

enumerator kCLOCK_Lpi2cClkRoot

LPI2C clock root.

enumerator kCLOCK_UartClkRoot

UART clock root.

enumerator kCLOCK_SpdifClkRoot

SPDIF clock root.

enumerator kCLOCK_Flexio1ClkRoot

FLEXIO1 clock root.

typedef enum _clock_name clock_name_t

Clock name used to get clock frequency.

typedef enum _clock_ip_name clock_ip_name_t

CCM CCGR gate control for each module independently.

typedef enum _clock_osc clock_osc_t

OSC 24M source select.

typedef enum _clock_gate_value clock_gate_value_t

Clock gate value.

typedef enum _clock_mode_t clock_mode_t

System clock mode.

typedef enum _clock_mux clock_mux_t

MUX control names for clock mux setting.

These constants define the mux control names for clock mux setting.

- 0:7: REG offset to CCM_BASE in bytes.
- 8:15: Root clock setting bit field shift.
- 16:31: Root clock setting bit field width.

typedef enum *_clock_div* clock_div_t

DIV control names for clock div setting.

These constants define div control names for clock div setting.

- 0:7: REG offset to CCM_BASE in bytes.
- 8:15: Root clock setting bit field shift.
- 16:31: Root clock setting bit field width.

typedef enum *_clock_div_value* clock_div_value_t

Clock divider value.

typedef enum *_clock_usb_src* clock_usb_src_t

USB clock source definition.

typedef enum *_clock_usb_phy_src* clock_usb_phy_src_t

Source of the USB HS PHY.

typedef struct *_clock_usb_pll_config* clock_usb_pll_config_t

PLL configuration for USB.

typedef struct *_clock_sys_pll_config* clock_sys_pll_config_t

PLL configuration for System.

typedef struct *_clock_audio_pll_config* clock_audio_pll_config_t

PLL configuration for AUDIO and VIDEO.

typedef struct *_clock_enet_pll_config* clock_enet_pll_config_t

PLL configuration for ENET.

typedef enum *_clock_pll* clock_pll_t

PLL name.

typedef enum *_clock_pfd* clock_pfd_t

PLL PFD name.

typedef enum *_clock_output1_selection* clock_output1_selection_t

The enumerator of clock output1's clock source, such as USB1 PLL, SYS PLL and so on.

typedef enum *_clock_output2_selection* clock_output2_selection_t

The enumerator of clock output2's clock source, such as USDHC1 clock root, LPI2C clock root and so on.

typedef enum *_clock_output_divider* clock_output_divider_t

The enumerator of clock output's divider.

typedef enum *_clock_root* clock_root_t

The enumerator of clock root.

volatile uint32_t g_xtalFreq

External XTAL (24M OSC/SYSOSC) clock frequency.

The XTAL (24M OSC/SYSOSC) clock frequency in Hz, when the clock is setup, use the function `CLOCK_SetXtalFreq` to set the value in to clock driver. For example, if XTAL is 24MHz,

```
CLOCK_InitExternalClk(false);  
CLOCK_SetXtalFreq(24000000);
```

volatile uint32_t g_rtcXtalFreq

External RTC XTAL (32K OSC) clock frequency.

The RTC XTAL (32K OSC) clock frequency in Hz, when the clock is setup, use the function `CLOCK_SetRtcXtalFreq` to set the value in to clock driver.

static inline void CLOCK_SetMux(*clock_mux_t* mux, uint32_t value)

Set CCM MUX node to certain value.

Parameters

- mux – Which mux node to set, see *clock_mux_t*.
- value – Clock mux value to set, different mux has different value range.

static inline uint32_t CLOCK_GetMux(*clock_mux_t* mux)

Get CCM MUX value.

Parameters

- mux – Which mux node to get, see *clock_mux_t*.

Returns

Clock mux value.

static inline void CLOCK_SetDiv(*clock_div_t* divider, uint32_t value)

Set clock divider value.

Example, set the ARM clock divider to divide by 2:

```
CLOCK_SetDiv(kCLOCK_ArmDiv, kCLOCK_ArmDivBy2);
```

Example, set the LPI2C clock divider to divide by 5.

```
CLOCK_SetDiv(kCLOCK_Lpi2cDiv, kCLOCK_MiscDivBy5);
```

Only kCLOCK_PerclkDiv, kCLOCK_UartDiv, kCLOCK_Sai3Div, kCLOCK_Sai1Div, kCLOCK_Lpi2cDiv can use the divider kCLOCK_MiscDivByxxx.

Parameters

- divider – Which divider node to set.
- value – Clock div value to set, different divider has different value range. See *clock_div_value_t* for details. Divided clock frequency = Undivided clock frequency / (value + 1)

static inline uint32_t CLOCK_GetDiv(*clock_div_t* divider)

Get CCM DIV node value.

Parameters

- divider – Which div node to get, see *clock_div_t*.

static inline void CLOCK_ControlGate(*clock_ip_name_t* name, *clock_gate_value_t* value)

Control the clock gate for specific IP.

Parameters

- name – Which clock to enable, see *clock_ip_name_t*.
- value – Clock gate value to set, see *clock_gate_value_t*.

static inline void CLOCK_EnableClock(*clock_ip_name_t* name)

Enable the clock for specific IP.

Parameters

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

static inline void CLOCK_DisableClock(*clock_ip_name_t* name)

Disable the clock for specific IP.

Parameters

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

static inline void CLOCK_SetMode(*clock_mode_t* mode)

Setting the low power mode that system will enter on next assertion of dsm_request signal.

Parameters

- mode – Which mode to enter, see clock_mode_t.

static inline uint32_t CLOCK_GetOscFreq(void)

Gets the OSC clock frequency.

This function will return the external XTAL OSC frequency if it is selected as the source of OSC, otherwise internal 24MHz RC OSC frequency will be returned.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetCoreFreq(void)

Gets the CORE clock frequency.

Returns

The CORE clock frequency value in hertz.

uint32_t CLOCK_GetIpgFreq(void)

Gets the IPG clock frequency.

Returns

The IPG clock frequency value in hertz.

uint32_t CLOCK_GetPerClkFreq(void)

Gets the PER clock frequency.

Returns

The PER clock frequency value in hertz.

uint32_t CLOCK_GetFreq(*clock_name_t* name)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in clock_name_t.

Parameters

- name – Clock names defined in clock_name_t

Returns

Clock frequency value in hertz

static inline uint32_t CLOCK_GetCpuClkFreq(void)

Get the CCM CPU/core/system frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetClockRootFreq(*clock_root_t* clockRoot)

Gets the frequency of selected clock root.

Parameters

- clockRoot – The clock root used to get the frequency, please refer to clock_root_t.

Returns

The frequency of selected clock root.

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.5.3.

CCM_ANALOG_PLL_BYPASS_SHIFT

CCM_ANALOG_PLL_BYPASS_SHIFT

CCM_ANALOG_PLL_BYPASS_CLK_SRC_MASK

CCM_ANALOG_PLL_BYPASS_CLK_SRC_MASK

CCM_ANALOG_PLL_BYPASS_CLK_SRC_SHIFT

CCM_ANALOG_PLL_BYPASS_CLK_SRC_SHIFT

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

CCSR_OFFSET

CCM registers offset.

CBCDR_OFFSET

CBCMR_OFFSET

CSCMR1_OFFSET

CSCMR2_OFFSET

CSCDR1_OFFSET

CDCDR_OFFSET

CSCDR2_OFFSET

CS1CDR_OFFSET

PLL_SYS_OFFSET

CCM Analog registers offset.

PLL_USB1_OFFSET

PLL_AUDIO_OFFSET

PLL_ENET_OFFSET

CCM_TUPLE(reg, shift, mask, busyShift)

CCM_TUPLE_REG(base, tuple)

CCM_TUPLE_SHIFT(tuple)

CCM_TUPLE_MASK(tuple)

CCM_TUPLE_BUSY_SHIFT(tuple)

CCM_NO_BUSY_WAIT

CCM_ANALOG_TUPLE(reg, shift)

CCM ANALOG tuple macros to map corresponding registers and bit fields.

CCM_ANALOG_TUPLE_SHIFT(tuple)

CCM_ANALOG_TUPLE_REG_OFF(base, tuple, off)

CCM_ANALOG_TUPLE_REG(base, tuple)

CLKPN_FREQ

clock1PN frequency.

CLOCK_SetXtal0Freq

CLOCK_SetXtal32Freq

ADC_CLOCKS

Clock ip name array for ADC.

AOI_CLOCKS

Clock ip name array for AOI.

DCDC_CLOCKS

Clock ip name array for DCDC.

DCP_CLOCKS

Clock ip name array for DCP.

DMAMUX_CLOCKS

Clock ip name array for DMAMUX_CLOCKS.

EDMA_CLOCKS

Clock ip name array for DMA.

EWM_CLOCKS

Clock ip name array for EWM.

FLEXIO_CLOCKS

Clock ip name array for FLEXIO.

FLEXRAM_CLOCKS

Clock ip name array for FLEXRAM.

FLEXSPI_CLOCKS

Clock ip name array for FLEXSPI.

GPIO_CLOCKS

Clock ip name array for GPIO.

GPT_CLOCKS

Clock ip name array for GPT.

KPP_CLOCKS

Clock ip name array for KPP.

LPI2C_CLOCKS

Clock ip name array for LPI2C.

LPSPi_CLOCKS

Clock ip name array for LPSPi.

LPUART_CLOCKS

Clock ip name array for LPUART.

OCRAM_EXSC_CLOCKS

Clock ip name array for OCRAM EXSC.

PIT_CLOCKS

Clock ip name array for PIT.

PWM_CLOCKS

Clock ip name array for PWM.

RTWDOG_CLOCKS

Clock ip name array for RTWDOG.

SAI_CLOCKS

Clock ip name array for SAI.

TRNG_CLOCKS

Clock ip name array for TRNG.

WDOG_CLOCKS

Clock ip name array for WDOG.

SPDIF_CLOCKS

Clock ip name array for SPDIF.

XBARA_CLOCKS

Clock ip name array for XBARA.

CLOCK_SOURCE_NONE

CLOCK_ROOT_SOUCE

CLOCK_ROOT_MUX_TUPLE

CLOCK_ROOT_NONE_PRE_DIV

CLOCK_ROOT_DIV_TUPLE

kCLOCK_CoreSysClk

For compatible with other platforms without CCM.

CLOCK_GetCoreSysClkFreq

For compatible with other platforms without CCM.

void CLOCK_InitExternalClk(bool bypassXtalOsc)

Initialize the external 24MHz clock.

This function supports two modes:

- a. Use external crystal oscillator.
- b. Bypass the external crystal oscillator, using input source clock directly.

After this function, please call CLOCK_SetXtal0Freq to inform clock driver the external clock frequency.

Note: This device does not support bypass external crystal oscillator, so the input parameter should always be false.

Parameters

- bypassXtalOsc – Pass in true to bypass the external crystal oscillator.

void CLOCK_DeinitExternalClk(void)

Deinitialize the external 24MHz clock.

This function disables the external 24MHz clock.

After this function, please call CLOCK_SetXtal0Freq to set external clock frequency to 0.

void CLOCK_SwitchOsc(clock_osc_t osc)

Switch the OSC.

This function switches the OSC source for SoC.

Parameters

- osc – OSC source to switch to.

static inline uint32_t CLOCK_GetRtcFreq(void)

Gets the RTC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline void CLOCK_SetXtalFreq(uint32_t freq)

Set the XTAL (24M OSC) frequency based on board setting.

Parameters

- freq – The XTAL input clock frequency in Hz.

static inline void CLOCK_SetRtcXtalFreq(uint32_t freq)

Set the RTC XTAL (32K OSC) frequency based on board setting.

Parameters

- freq – The RTC XTAL input clock frequency in Hz.

void CLOCK_InitRcOsc24M(void)

Initialize the RC oscillator 24MHz clock.

void CLOCK_DeinitRcOsc24M(void)

Power down the RCOSC 24M clock.

bool CLOCK_EnableUsbhs0Clock(*clock_usb_src_t* src, uint32_t freq)

Enable USB HS clock.

This function only enables the access to USB HS peripheral, upper layer should first call the `CLOCK_EnableUsbhs0PhyPllClock` to enable the PHY clock to use USB HS.

Parameters

- src – USB HS does not care about the clock source, here must be `kCLOCK_UsbSrcUnused`.
- freq – USB HS does not care about the clock source, so this parameter is ignored.

Return values

- true – The clock is set successfully.
- false – The clock source is invalid to get proper USB HS clock.

static inline void CLOCK_SetPllBypass(CCM_ANALOG_Type *base, *clock_pll_t* pll, bool bypass)

PLL bypass setting.

Parameters

- base – CCM_ANALOG base pointer.
- pll – PLL control name (see `ccm_analog_pll_control_t` enumeration)
- bypass – Bypass the PLL.
 - true: Bypass the PLL.
 - false: Not bypass the PLL.

static inline bool CLOCK_IsPllBypassed(CCM_ANALOG_Type *base, *clock_pll_t* pll)

Check if PLL is bypassed.

Parameters

- base – CCM_ANALOG base pointer.
- pll – PLL control name (see `ccm_analog_pll_control_t` enumeration)

Returns

PLL bypass status.

- true: The PLL is bypassed.
- false: The PLL is not bypassed.

static inline bool CLOCK_IsPllEnabled(CCM_ANALOG_Type *base, *clock_pll_t* pll)

Check if PLL is enabled.

Parameters

- base – CCM_ANALOG base pointer.
- pll – PLL control name (see *ccm_analog_pll_control_t* enumeration)

Returns

PLL bypass status.

- true: The PLL is enabled.
- false: The PLL is not enabled.

static inline void CLOCK_SetPllBypassRefClkSrc(CCM_ANALOG_Type *base, *clock_pll_t* pll, *uint32_t* src)

PLL bypass clock source setting. Note: change the bypass clock source also change the pll reference clock source.

Parameters

- base – CCM_ANALOG base pointer.
- pll – PLL control name (see *ccm_analog_pll_control_t* enumeration)
- src – Bypass clock source, reference *_clock_pll_bypass_clk_src*.

static inline *uint32_t* CLOCK_GetPllBypassRefClk(CCM_ANALOG_Type *base, *clock_pll_t* pll)

Get PLL bypass clock value, it is PLL reference clock actually. If CLOCK1_P, CLOCK1_N is choose as the pll bypass clock source, please implement the CLKPN_FREQ define, otherwise 0 will be returned.

Parameters

- base – CCM_ANALOG base pointer.
- pll – PLL control name (see *ccm_analog_pll_control_t* enumeration)

Return values

bypass – reference clock frequency value.

void CLOCK_InitSysPll(const *clock_sys_pll_config_t* *config)

Initialize the System PLL.

This function initializes the System PLL with specific settings

Parameters

- config – Configuration to set to PLL.

void CLOCK_DeinitSysPll(void)

De-initialize the System PLL.

void CLOCK_InitUsb1Pll(const *clock_usb_pll_config_t* *config)

Initialize the USB1 PLL.

This function initializes the USB1 PLL with specific settings

Parameters

- config – Configuration to set to PLL.

void CLOCK_DeinitUsb1Pll(void)

Deinitialize the USB1 PLL.

void CLOCK_InitAudioPll(const *clock_audio_pll_config_t* *config)

Initializes the Audio PLL.

This function initializes the Audio PLL with specific settings

Parameters

- config – Configuration to set to PLL.

void CLOCK_DeinitAudioPll(void)

De-initialize the Audio PLL.

void CLOCK_InitEnetPll(const *clock_enet_pll_config_t* *config)

Initialize the ENET PLL.

This function initializes the ENET PLL with specific settings.

Parameters

- config – Configuration to set to PLL.

void CLOCK_DeinitEnetPll(void)

Deinitialize the ENET PLL.

This function disables the ENET PLL.

uint32_t CLOCK_GetPllFreq(*clock_pll_t* pll)

Get current PLL output frequency.

This function get current output frequency of specific PLL

Parameters

- pll – pll name to get frequency.

Returns

The PLL output frequency in hertz.

void CLOCK_InitSysPfd(*clock_pfd_t* pfd, uint8_t pfdFrac)

Initialize the System PLL PFD.

This function initializes the System PLL PFD. During new value setting, the clock output is disabled to prevent glitch.

Note: It is recommended that PFD settings are kept between 12-35.

Parameters

- pfd – Which PFD clock to enable.
- pfdFrac – The PFD FRAC value.

void CLOCK_DeinitSysPfd(*clock_pfd_t* pfd)

De-initialize the System PLL PFD.

This function disables the System PLL PFD.

Parameters

- pfd – Which PFD clock to disable.

bool CLOCK_IsSysPfdEnabled(*clock_pfd_t* pfd)

Check if Sys PFD is enabled.

Parameters

- pfd – PFD control name

Returns

PFD bypass status.

- true: power on.
- false: power off.

void CLOCK_InitUsb1Pfd(*clock_pfd_t* pfd, uint8_t pfdFrac)

Initialize the USB1 PLL PFD.

This function initializes the USB1 PLL PFD. During new value setting, the clock output is disabled to prevent glitch.

Note: It is recommended that PFD settings are kept between 12-35.

Parameters

- pfd – Which PFD clock to enable.
- pfdFrac – The PFD FRAC value.

void CLOCK_DeinitUsb1Pfd(*clock_pfd_t* pfd)

De-initialize the USB1 PLL PFD.

This function disables the USB1 PLL PFD.

Parameters

- pfd – Which PFD clock to disable.

bool CLOCK_IsUsb1PfdEnabled(*clock_pfd_t* pfd)

Check if Usb1 PFD is enabled.

Parameters

- pfd – PFD control name.

Returns

PFD bypass status.

- true: power on.
- false: power off.

uint32_t CLOCK_GetSysPfdFreq(*clock_pfd_t* pfd)

Get current System PLL PFD output frequency.

This function get current output frequency of specific System PLL PFD

Parameters

- pfd – pfd name to get frequency.

Returns

The PFD output frequency in hertz.

uint32_t CLOCK_GetUsb1PfdFreq(*clock_pfd_t* pfd)

Get current USB1 PLL PFD output frequency.

This function get current output frequency of specific USB1 PLL PFD

Parameters

- pfd – pfd name to get frequency.

Returns

The PFD output frequency in hertz.

`bool CLOCK_EnableUsbhs0PhyPllClock(clock_usb_phy_src_t src, uint32_t freq)`

Enable USB HS PHY PLL clock.

This function enables the internal 480MHz USB PHY PLL clock.

Parameters

- src – USB HS PHY PLL clock source.
- freq – The frequency specified by src.

Return values

- true – The clock is set successfully.
- false – The clock source is invalid to get proper USB HS clock.

`void CLOCK_DisableUsbhs0PhyPllClock(void)`

Disable USB HS PHY PLL clock.

This function disables USB HS PHY PLL clock.

`void CLOCK_SetClockOutput1(clock_output1_selection_t selection, clock_output_divider_t divider)`

Set the clock source and the divider of the clock output1.

Parameters

- selection – The clock source to be output, please refer to `clock_output1_selection_t`.
- divider – The divider of the output clock signal, please refer to `clock_output_divider_t`.

`void CLOCK_SetClockOutput2(clock_output2_selection_t selection, clock_output_divider_t divider)`

Set the clock source and the divider of the clock output2.

Parameters

- selection – The clock source to be output, please refer to `clock_output2_selection_t`.
- divider – The divider of the output clock signal, please refer to `clock_output_divider_t`.

`uint32_t CLOCK_GetClockOutCLKO1Freq(void)`

Get the frequency of clock output1 clock signal.

Returns

The frequency of clock output1 clock signal.

`uint32_t CLOCK_GetClockOutClkO2Freq(void)`

Get the frequency of clock output2 clock signal.

Returns

The frequency of clock output2 clock signal.

`FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL`

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

struct `_clock_usb_pll_config`
#include <fsl_clock.h> PLL configuration for USB.

Public Members

uint8_t loopDivider
PLL loop divider. 0 - Fout=Fref*20; 1 - Fout=Fref*22

uint8_t src
Pll clock source, reference `_clock_pll_clk_src`

struct `_clock_sys_pll_config`
#include <fsl_clock.h> PLL configuration for System.

Public Members

uint8_t loopDivider
PLL loop divider. Intended to be 1 (528M). 0 - Fout=Fref*20; 1 - Fout=Fref*22

uint32_t numerator
30 bit numerator of fractional loop divider.

uint32_t denominator
30 bit denominator of fractional loop divider

uint8_t src
Pll clock source, reference `_clock_pll_clk_src`

uint16_t ss_stop
Stop value to get frequency change.

uint8_t ss_enable
Enable spread spectrum modulation

uint16_t ss_step
Step value to get frequency change step.

struct `_clock_audio_pll_config`
#include <fsl_clock.h> PLL configuration for AUDIO and VIDEO.

Public Members

uint8_t loopDivider
PLL loop divider. Valid range for DIV_SELECT divider value: 27~54.

uint8_t postDivider
Divider after the PLL, should only be 1, 2, 4, 8, 16.

uint32_t numerator
30 bit numerator of fractional loop divider.

uint32_t denominator
30 bit denominator of fractional loop divider

```
uint8_t src
    Pll clock source, reference _clock_pll_clk_src
struct _clock_enet_pll_config
    #include <fsl_clock.h> PLL configuration for ENET.
```

Public Members

```
bool enableClkOutput
    Power on and enable PLL clock output for ENET0 (ref_enetpll0).
bool enableClkOutput500M
    Power on and enable PLL clock output for ENET (ref_enetpll500M).
bool enableClkOutput25M
    Power on and enable PLL clock output for ENET1 (ref_enetpll1).
uint8_t loopDivider
    Controls the frequency of the ENET0 reference clock. b00 25MHz b01 50MHz b10
    100MHz (not 50% duty cycle) b11 125MHz
uint8_t src
    Pll clock source, reference _clock_pll_clk_src
```

2.7 DCDC: DCDC Converter

```
void DCDC_Init(DCDC_Type *base, dcdc_config_t *config)
    Enable the access to DCDC registers.
```

Parameters

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

```
void DCDC_Deinit(DCDC_Type *base)
    Disable the access to DCDC registers.
```

Parameters

- base – DCDC peripheral base address.

```
void DCDC_GetDefaultConfig(DCDC_Type *base, dcdc_config_t *config)
    brief Get the default setting for DCDC user configuration structure.
```

This function initializes the user configuration structure to a default value. The default values are: code config->controlMode = kDCDC_StaticControl; config->trimInputMode = kDCDC_SampleTrimInput; config->enableDcdcTimeout = false; config->enableSwitchingConverterOutput = false; endcode

param config Pointer to configuration structure. See to “dcdc_config_t”

```
uint32_t DCDC_GetstatusFlags(DCDC_Type *base)
    Get DCDC status flags.
```

Parameters

- base – peripheral base address.

Returns

Mask of asserted status flags. See to “_dcdc_status_flags_t”.

void DCDC_EnterLowPowerMode(DCDC_Type *base, *dcdc_low_power_mode_t* mode)

Make DCDC enter into low power modes.

Parameters

- base – DCDC peripheral base address.
- mode – DCDC low power mode selection. See to “*_dcdc_low_power_mode*”

static inline void DCDC_EnableOutputRangeComparator(DCDC_Type *base, bool enable)

Enable the output range comparator.

The output range comparator is disabled by default.

Parameters

- base – DCDC peripheral base address.
- enable – Enable the feature or not.

void DCDC_SetClockSource(DCDC_Type *base, *dcdc_clock_source_t* clockSource)

Configure the DCDC clock source.

Parameters

- base – DCDC peripheral base address.
- clockSource – Clock source for DCDC. See to “*dcdc_clock_source_t*”.

void DCDC_GetDefaultDetectionConfig(*dcdc_detection_config_t* *config)

Get the default setting for detection configuration.

The default configuration are set according to responding registers' setting when powered on. They are:

```
config->enableXtalokDetection = false;
config->powerDownOverVoltageDetection = true;
config->powerDownLowVoltageDetection = false;
config->powerDownOverCurrentDetection = true;
config->powerDownPeakCurrentDetection = true;
config->powerDownZeroCrossDetection = true;
config->OverCurrentThreshold = kDCDC_OverCurrentThresholdAlt0;
config->PeakCurrentThreshold = kDCDC_PeakCurrentThresholdAlt0;
```

Parameters

- config – Pointer to configuration structure. See to “*dcdc_detection_config_t*”

void DCDC_SetDetectionConfig(DCDC_Type *base, const *dcdc_detection_config_t* *config)

Configure the DCDC detection.

Parameters

- base – DCDC peripheral base address.
- config – Pointer to configuration structure. See to “*dcdc_detection_config_t*”

void DCDC_GetDefaultLowPowerConfig(*dcdc_low_power_config_t* *config)

Get the default setting for low power configuration.

The default configuration are set according to responding registers' setting when powered on. They are:

```
config->enableOverloadDetection = true;
config->enableAdjustHystereticValue = false;
config->countChargingTimePeriod = kDCDC_CountChargingTimePeriod8Cycle;
config->countChargingTimeThreshold = kDCDC_CountChargingTimeThreshold32;
```

Parameters

- config – Pointer to configuration structure. See to “*dcdc_low_power_config_t*”

void DCDC_SetLowPowerConfig(DCDC_Type *base, const *dcdc_low_power_config_t* *config)

Configure the DCDC low power.

Parameters

- base – DCDC peripheral base address.
- config – Pointer to configuration structure. See to “*dcdc_low_power_config_t*”.

void DCDC_ResetCurrentAlertSignal(DCDC_Type *base, bool enable)

Reset current alert signal. Alert signal is generate by peak current detection.

Parameters

- base – DCDC peripheral base address.
- enable – Switcher to reset signal. True means reset signal. False means don't reset signal.

static inline void DCDC_SetBandgapVoltageTrimValue(DCDC_Type *base, uint32_t trimValue)

Set the bangap trim value to trim bandgap voltage.

Parameters

- base – DCDC peripheral base address.
- trimValue – The bangap trim value. Available range is 0U-31U.

void DCDC_GetDefaultLoopControlConfig(*dcdc_loop_control_config_t* *config)

Get the default setting for loop control configuration.

The default configuration are set according to responding registers' setting when powered on. They are:

```
config->enableCommonHysteresis = false;
config->enableCommonThresholdDetection = false;
config->enableInvertHysteresisSign = false;
config->enableRCThresholdDetection = false;
config->enableRCScaleCircuit = 0U;
config->complementFeedForwardStep = 0U;
```

Parameters

- config – Pointer to configuration structure. See to “*dcdc_loop_control_config_t*”

void DCDC_SetLoopControlConfig(DCDC_Type *base, const *dcdc_loop_control_config_t* *config)

Configure the DCDC loop control.

Parameters

- base – DCDC peripheral base address.
- config – Pointer to configuration structure. See to “*dcdc_loop_control_config_t*”.

```
void DCDC_SetMinPowerConfig(DCDC_Type *base, const dcdc_min_power_config_t *config)
```

Configure for the min power.

Parameters

- base – DCDC peripheral base address.
- config – Pointer to configuration structure. See to “*dcdc_min_power_config_t*”.

```
static inline void DCDC_SetLPComparatorBiasValue(DCDC_Type *base,  
                                                  dcdc_comparator_current_bias_t biasVaule)
```

Set the current bias of low power comparator.

Parameters

- base – DCDC peripheral base address.
- biasVaule – The current bias of low power comparator. Refer to “*dcdc_comparator_current_bias_t*”.

```
static inline void DCDC_LockTargetVoltage(DCDC_Type *base)
```

Lock target voltage.

Parameters

- base – DCDC peripheral base address.

```
void DCDC_AdjustTargetVoltage(DCDC_Type *base, uint32_t VDDRun, uint32_t VDDStandby)
```

Adjust the target voltage of VDD_SOC in run mode and low power mode.

Deprecated:

Do not use this function. It has been supersceded by `DCDC_AdjustRunTargetVoltage` and `DCDC_AdjustLowPowerTargetVoltage`

This function is to adjust the target voltage of DCDC output. Change them and finally wait until the output is stabled. Set the target value of run mode the same as low power mode before entering power save mode, because DCDC will switch back to run mode if it detects the current loading is larger than about 50 mA(typical value).

Parameters

- base – DCDC peripheral base address.
- VDDRun – Target value in run mode. 25 mV each step from 0x00 to 0x1F. 00 is for 0.8V, 0x1F is for 1.575V.
- VDDStandby – Target value in low power mode. 25 mV each step from 0x00 to 0x4. 00 is for 0.9V, 0x4 is for 1.0V.

```
void DCDC_AdjustRunTargetVoltage(DCDC_Type *base, uint32_t VDDRun)
```

Adjust the target voltage of VDD_SOC in run mode.

This function is to adjust the target voltage of DCDC output. Change them and finally wait until the output is stabled. Set the target value of run mode the same as low power mode before entering power save mode, because DCDC will switch back to run mode if it detects the current loading is larger than about 50 mA(typical value).

Parameters

- base – DCDC peripheral base address.
- VDDRun – Target value in run mode. 25 mV each step from 0x00 to 0x1F. 00 is for 0.8V, 0x1F is for 1.575V.

`void DCDC_AdjustLowPowerTargetVoltage(DCDC_Type *base, uint32_t VDDStandby)`

Adjust the target voltage of VDD_SOC in low power mode.

This function is to adjust the target voltage of DCDC output. Change them and finally wait until the output is stabled. Set the target value of run mode the same as low power mode before entering power save mode, because DCDC will switch back to run mode if it detects the current loading is larger than about 50 mA(typical value).

Parameters

- base – DCDC peripheral base address.
- VDDStandby – Target value in low power mode. 25 mV each step from 0x00 to 0x4. 00 is for 0.9V, 0x4 is for 1.0V.

`void DCDC_SetInternalRegulatorConfig(DCDC_Type *base, const
dcdc_internal_regulator_config_t *config)`

Configure the DCDC internal regulator.

Parameters

- base – DCDC peripheral base address.
- config – Pointer to configuration structure. See to “`dcdc_internal_regulator_config_t`”.

`static inline void DCDC_EnableImproveTransition(DCDC_Type *base, bool enable)`

Enable/Disable to improve the transition from heavy load to light load. It is valid while zero cross detection is enabled. If output exceeds the threshold, DCDC would return CCM from DCM.

Parameters

- base – DCDC peripheral base address.
- enable – Enable the feature or not.

`void DCDC_BootIntoDCM(DCDC_Type *base)`

Boot DCDC into DCM(discontinuous conduction mode).

`pwd_zcd=0x0; pwd_cmp_offset=0x0; dcdc_loopctrl_en_rcscale= 0x5; DCM_set_ctrl=1'b1;`

Parameters

- base – DCDC peripheral base address.

`void DCDC_BootIntoCCM(DCDC_Type *base)`

Boot DCDC into CCM(continous conduction mode).

`pwd_zcd=0x1; pwd_cmp_offset=0x0; dcdc_loopctrl_en_rcscale=0x3;`

Parameters

- base – DCDC peripheral base address.

`enum _dcdc_status_flags_t`

DCDC status flags.

Values:

enumerator `kDCDC_LockedOKStatus`

Indicate DCDC status. 1'b1: DCDC already settled 1'b0: DCDC is settling.

`enum _dcdc_comparator_current_bias`

The current bias of low power comparator.

Values:

enumerator kDCDC_ComparatorCurrentBias50nA

The current bias of low power comparator is 50nA.

enumerator kDCDC_ComparatorCurrentBias100nA

The current bias of low power comparator is 100nA.

enumerator kDCDC_ComparatorCurrentBias200nA

The current bias of low power comparator is 200nA.

enumerator kDCDC_ComparatorCurrentBias400nA

The current bias of low power comparator is 400nA.

enum _dcdc_over_current_threshold

The threshold of over current detection.

Values:

enumerator kDCDC_OverCurrentThresholdAlt0

1A in the run mode, 0.25A in the power save mode.

enumerator kDCDC_OverCurrentThresholdAlt1

2A in the run mode, 0.25A in the power save mode.

enumerator kDCDC_OverCurrentThresholdAlt2

1A in the run mode, 0.2A in the power save mode.

enumerator kDCDC_OverCurrentThresholdAlt3

2A in the run mode, 0.2A in the power save mode.

enum _dcdc_peak_current_threshold

The threshold if peak current detection.

Values:

enumerator kDCDC_PeakCurrentThresholdAlt0

150mA peak current threshold.

enumerator kDCDC_PeakCurrentThresholdAlt1

250mA peak current threshold.

enumerator kDCDC_PeakCurrentThresholdAlt2

350mA peak current threshold.

enumerator kDCDC_PeakCurrentThresholdAlt3

450mA peak current threshold.

enumerator kDCDC_PeakCurrentThresholdAlt4

550mA peak current threshold.

enumerator kDCDC_PeakCurrentThresholdAlt5

650mA peak current threshold.

enum _dcdc_count_charging_time_period

The period of counting the charging times in power save mode.

Values:

enumerator kDCDC_CountChargingTimePeriod8Cycle

Eight 32k cycle.

enumerator kDCDC_CountChargingTimePeriod16Cycle

Sixteen 32k cycle.

enum _dcdc_count_charging_time_threshold

The threshold of the counting number of charging times.

Values:

enumerator kDCDC_CountChargingTimeThreshold32

0x0: 32.

enumerator kDCDC_CountChargingTimeThreshold64

0x1: 64.

enumerator kDCDC_CountChargingTimeThreshold16

0x2: 16.

enumerator kDCDC_CountChargingTimeThreshold8

0x3: 8.

enum _dcdc_clock_source

Oscillator clock option.

Values:

enumerator kDCDC_ClockAutoSwitch

Automatic clock switch from internal oscillator to external clock.

enumerator kDCDC_ClockInternalOsc

Use internal oscillator.

enumerator kDCDC_ClockExternalOsc

Use external 24M crystal oscillator.

enum _dcdc_low_power_mode

DCDC low power modes.

Values:

enumerator kDCDC_StandbyMode

Standby mode.

enumerator kDCDC_LowPowerMode

Low power mode.

enumerator kDCDC_GpcStandbyLowPowerMode

low power mode for GPC standby request.

enum _dcdc_control_mode

DCDC control mode.

Values:

enumerator kDCDC_StaticControl

Static control.

enumerator kDCDC_SetPointControl

Controlled by GPC set points.

enum _dcdc_trim_input_mode

DCDC trim input mode.

Values:

enumerator kDCDC_SampleTrimInput

Sample trim input.

enumerator `kDCDC_HoldTrimInput`

Hold trim input.

typedef enum `_dcdc_comparator_current_bias` `dcdc_comparator_current_bias_t`

The current bias of low power comparator.

typedef enum `_dcdc_over_current_threshold` `dcdc_over_current_threshold_t`

The threshold of over current detection.

typedef enum `_dcdc_peak_current_threshold` `dcdc_peak_current_threshold_t`

The threshold if peak current detection.

typedef enum `_dcdc_count_charging_time_period` `dcdc_count_charging_time_period_t`

The period of counting the charging times in power save mode.

typedef enum `_dcdc_count_charging_time_threshold` `dcdc_count_charging_time_threshold_t`

The threshold of the counting number of charging times.

typedef enum `_dcdc_clock_source` `dcdc_clock_source_t`

Oscillator clock option.

typedef enum `_dcdc_low_power_mode` `dcdc_low_power_mode_t`

DCDC low power modes.

typedef enum `_dcdc_control_mode` `dcdc_control_mode_t`

DCDC control mode.

typedef enum `_dcdc_trim_input_mode` `dcdc_trim_input_mode_t`

DCDC trim input mode.

typedef struct `_dcdc_config` `dcdc_config_t`

Configuration for DCDC.

typedef struct `_dcdc_detection_config` `dcdc_detection_config_t`

Configuration for DCDC detection.

typedef struct `_dcdc_loop_control_config` `dcdc_loop_control_config_t`

Configuration for the loop control.

typedef struct `_dcdc_low_power_config` `dcdc_low_power_config_t`

Configuration for DCDC low power.

typedef struct `_dcdc_internal_regulator_config` `dcdc_internal_regulator_config_t`

Configuration for DCDC internal regulator.

typedef struct `_dcdc_min_power_config` `dcdc_min_power_config_t`

Configuration for min power setting.

FSL_DCDC_DRIVER_VERSION

DCDC driver version.

Version 2.3.0.

struct `_dcdc_config`

`#include <fsl_dcdc.h>` Configuration for DCDC.

Public Members

`dcdc_control_mode_t` controlMode

DCDC control mode.

dcdc_trim_input_mode_t trimInputMode

Hold trim input.

bool enableDcdcTimeout

Enable internal count for DCDC_OK timeout.

bool enableSwitchingConverterOutput

Enable the VDDIO switching converter output.

struct __dcdc_detection_config

#include <fsl_dcdc.h> Configuration for DCDC detection.

Public Members

bool enableXtalokDetection

Enable xtalog detection circuit.

bool powerDownOverVoltageDetection

Power down over-voltage detection comparator.

bool powerDownLowVoltageDetection

Power down low-voltage detection comparator.

bool powerDownOverCurrentDetection

Power down over-current detection.

bool powerDownPeakCurrentDetection

Power down peak-current detection.

bool powerDownZeroCrossDetection

Power down the zero cross detection function for discontinuous conductor mode.

dcdc_over_current_threshold_t OverCurrentThreshold

The threshold of over current detection.

dcdc_peak_current_threshold_t PeakCurrentThreshold

The threshold of peak current detection.

struct __dcdc_loop_control_config

#include <fsl_dcdc.h> Configuration for the loop control.

Public Members

bool enableCommonHysteresis

Enable hysteresis in switching converter common mode analog comparators. This feature will improve transient supply ripple and efficiency.

bool enableCommonThresholdDetection

Increase the threshold detection for common mode analog comparator.

bool enableDifferentialHysteresis

Enable hysteresis in switching converter differential mode analog comparators. This feature will improve transient supply ripple and efficiency.

bool enableDifferentialThresholdDetection

Increase the threshold detection for differential mode analog comparators.

bool enableInvertHysteresisSign

Invert the sign of the hysteresis in DC-DC analog comparators.

bool enableRCThresholdDetection

Increase the threshold detection for RC scale circuit.

uint32_t enableRCScaleCircuit

Available range is 0~7. Enable analog circuit of DC-DC converter to respond faster under transient load conditions.

uint32_t complementFeedForwardStep

Available range is 0~7. Two's complement feed forward step in duty cycle in the switching DC-DC converter. Each time this field makes a transition from 0x0, the loop filter of the DC-DC converter is stepped once by a value proportional to the change. This can be used to force a certain control loop behavior, such as improving response under known heavy load transients.

struct __dcdc_low_power_config

#include <fsl_dcdc.h> Configuration for DCDC low power.

Public Members

bool enableAdjustHystereticValue

Adjust hysteretic value in low power from 12.5mV to 25mV.

dcdc_count_charging_time_period_t countChargingTimePeriod

The period of counting the charging times in power save mode.

dcdc_count_charging_time_threshold_t countChargingTimeThreshold

the threshold of the counting number of charging times during the period that lp_overload_freq_sel sets in power save mode.

struct __dcdc_internal_regulator_config

#include <fsl_dcdc.h> Configuration for DCDC internal regulator.

Public Members

bool enableLoadResistor

control the load resistor of the internal regulator of DCDC, the load resistor is connected as default “true”, and need set to “false” to disconnect the load resistor.

uint32_t feedbackPoint

Available range is 0~3. Select the feedback point of the internal regulator.

struct __dcdc_min_power_config

#include <fsl_dcdc.h> Configuration for min power setting.

Public Members

bool enableUseHalfFreqForContinuous

Set DCDC clock to half frequency for the continuous mode.

2.8 DCP: Data Co-Processor

FSL_DCP_DRIVER_VERSION

DCP driver version. Version 2.1.7.

Current version: 2.1.7

Change log:

- Version 2.1.7
 - Bug Fix
 - * Reduce optimization level for critical functions working with SRF.
- Version 2.1.6
 - Bug Fix
 - * MISRA C-2012 issue fix.
- Version 2.1.5
 - Improvements
 - * Add support for DCACHE.
- Version 2.1.4
 - Bug Fix
 - * Fix CRC-32 computation issue on the code's block boundary size.
- Version 2.1.3
 - Bug Fix
 - * MISRA C-2012 issue fixed: rule 10.1, 10.3, 10.4, 11.9, 14.4, 16.4 and 17.7.
- Version 2.1.2
 - Fix sign-compare warning in dcp_reverse_and_copy.
- Version 2.1.1
 - Add DCP status clearing when channel operation is complete
- 2.1.0
 - Add byte/word swap feature for key, input and output data
- Version 2.0.0
 - Initial version

enum _dcp_status

DCP status return codes.

Values:

enumerator kStatus_DCP_Again

Non-blocking function shall be called again.

enum _dcp_ch_enable

DCP channel enable.

Values:

enumerator kDCP_chDisable

DCP channel disable

enumerator kDCP_ch0Enable

DCP channel 0 enable

enumerator kDCP_ch1Enable

DCP channel 1 enable

enumerator kDCP_ch2Enable

DCP channel 2 enable

enumerator kDCP_ch3Enable

DCP channel 3 enable

enumerator kDCP_chEnableAll

DCP channel enable all

enum _dcp_ch_int_enable

DCP interrupt enable.

Values:

enumerator kDCP_chIntDisable

DCP interrupts disable

enumerator kDCP_ch0IntEnable

DCP channel 0 interrupt enable

enumerator kDCP_ch1IntEnable

DCP channel 1 interrupt enable

enumerator kDCP_ch2IntEnable

DCP channel 2 interrupt enable

enumerator kDCP_ch3IntEnable

DCP channel 3 interrupt enable

enum _dcp_channel

DCP channel selection.

Values:

enumerator kDCP_Channel0

DCP channel 0.

enumerator kDCP_Channel1

DCP channel 1.

enumerator kDCP_Channel2

DCP channel 2.

enumerator kDCP_Channel3

DCP channel 3.

enum _dcp_key_slot

DCP key slot selection.

Values:

enumerator kDCP_KeySlot0

DCP key slot 0.

enumerator kDCP_KeySlot1

DCP key slot 1.

enumerator kDCP_KeySlot2

DCP key slot 2.

enumerator kDCP_KeySlot3

DCP key slot 3.

enumerator kDCP_OtpKey

DCP OTP key.

enumerator kDCP_OtpUniqueKey

DCP unique OTP key.

enumerator kDCP_PayloadKey

DCP payload key.

enum _dcp_swap

DCP key, input & output swap options.

Values:

enumerator kDCP_NoSwap

enumerator kDCP_KeyByteSwap

enumerator kDCP_KeyWordSwap

enumerator kDCP_InputByteSwap

enumerator kDCP_InputWordSwap

enumerator kDCP_OutputByteSwap

enumerator kDCP_OutputWordSwap

typedef enum _dcp_ch_enable _dcp_ch_enable_t

DCP channel enable.

typedef enum _dcp_ch_int_enable _dcp_ch_int_enable_t

DCP interrupt enable.

typedef enum _dcp_channel dcp_channel_t

DCP channel selection.

typedef enum _dcp_key_slot dcp_key_slot_t

DCP key slot selection.

typedef enum _dcp_swap dcp_swap_t

DCP key, input & output swap options.

typedef struct _dcp_work_packet dcp_work_packet_t

DCP's work packet.

typedef struct _dcp_handle dcp_handle_t

Specify DCP's key resource and DCP channel.

typedef struct _dcp_context dcp_context_t

DCP's context buffer, used by DCP for context switching between channels.

typedef struct _dcp_config dcp_config_t

DCP's configuration structure.

void DCP_Init(DCP_Type *base, const dcp_config_t *config)

Enables clock to and enables DCP.

Enable DCP clock and configure DCP.

Parameters

- base – DCP base address
- config – Pointer to configuration structure.

void DCP_Deinit(DCP_Type *base)

Disable DCP clock.

Reset DCP and Disable DCP clock.

Parameters

- base – DCP base address

void DCP_GetDefaultConfig(*dcp_config_t* *config)

Gets the default configuration structure.

This function initializes the DCP configuration structure to a default value. The default values are as follows. dcpConfig->gatherResidualWrites = true; dcpConfig->enableContextCaching = true; dcpConfig->enableContextSwitching = true; dcpConfig->enableChannel = kDCP_chEnableAll; dcpConfig->enableChannelInterrupt = kDCP_chIntDisable;

Parameters

- config – **[out]** Pointer to configuration structure.

status_t DCP_WaitForChannelComplete(DCP_Type *base, *dcp_handle_t* *handle)

Poll and wait on DCP channel.

Polls the specified DCP channel until current it completes activity.

Parameters

- base – DCP peripheral base address.
- handle – Specifies DCP channel.

Returns

kStatus_Success When data processing completes without error.

Returns

kStatus_Fail When error occurs.

struct __dcp_work_packet

#include <fsl_dcp.h> DCP's work packet.

struct __dcp_handle

#include <fsl_dcp.h> Specify DCP's key resource and DCP channel.

Public Members

dcp_channel_t channel

Specify DCP channel.

dcp_key_slot_t keySlot

For operations with key (such as AES encryption/decryption), specify DCP key slot.

uint32_t swapConfig

For configuration of key, input, output byte/word swap options

struct __dcp_context

#include <fsl_dcp.h> DCP's context buffer, used by DCP for context switching between channels.

struct __dcp_config

#include <fsl_dcp.h> DCP's configuration structure.

Public Members

`bool` gatherResidualWrites

Enable the ragged writes to the unaligned buffers.

`bool` enableContextCaching

Enable the caching of contexts between the operations.

`bool` enableContextSwitching

Enable automatic context switching for the channels.

`uint8_t` enableChannel

DCP channel enable.

`uint8_t` enableChannelInterrupt

Per-channel interrupt enable.

2.9 DCP AES blocking driver

`status_t` DCP_AES_SetKey(DCP_Type *base, *dcp_handle_t* *handle, const `uint8_t` *key, `size_t` keySize)

Set AES key to *dcp_handle_t* struct and optionally to DCP.

Sets the AES key for encryption/decryption with the *dcp_handle_t* structure. The *dcp_handle_t* input argument specifies keySlot. If the keySlot is `kDCP_OtpKey`, the function will check the `OTP_KEY_READY` bit and will return it's ready to use status. For other keySlot selections, the function will copy and hold the key in *dcp_handle_t* struct. If the keySlot is one of the four DCP SRAM-based keys (one of `kDCP_KeySlot0`, `kDCP_KeySlot1`, `kDCP_KeySlot2`, `kDCP_KeySlot3`), this function will also load the supplied key to the specified keySlot in DCP.

Parameters

- base – DCP 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.

Returns

status from set key operation

`status_t` DCP_AES_EncryptEcb(DCP_Type *base, *dcp_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 – DCP 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 DCP_AES_DecryptEcb(DCP_Type *base, dcp_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 – DCP 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 DCP_AES_EncryptCbc(DCP_Type *base, dcp_handle_t *handle, const uint8_t
                             *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])
```

Encrypts AES using CBC block mode.

Encrypts AES using CBC block mode. The source plaintext and destination ciphertext can overlap in system memory.

Parameters

- base – DCP 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 DCP_AES_DecryptCbc(DCP_Type *base, dcp_handle_t *handle, const uint8_t
                             *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])
```

Decrypts AES using CBC block mode.

Decrypts AES using CBC block mode. The source ciphertext and destination plaintext can overlap in system memory.

Parameters

- base – DCP 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

DCP_AES_BLOCK_SIZE
AES block size in bytes

2.10 DCP HASH driver

enum _dcp_hash_algo_t

Supported cryptographic block cipher functions for HASH creation.

Values:

enumerator kDCP_Sha1

SHA_1

enumerator kDCP_Sha256

SHA_256

enumerator kDCP_Crc32

CRC_32

typedef enum _dcp_hash_algo_t dcp_hash_algo_t

Supported cryptographic block cipher functions for HASH creation.

typedef struct _dcp_hash_ctx_t dcp_hash_ctx_t

Storage type used to save hash context.

status_t DCP_HASH_Init(DCP_Type *base, dcp_handle_t *handle, dcp_hash_ctx_t *ctx,
dcp_hash_algo_t algo)

Initialize HASH context.

This function initializes the HASH.

Parameters

- base – DCP peripheral base address
- handle – Specifies the DCP channel used for hashing.
- ctx – **[out]** Output hash context
- algo – Underlying algorithm to use for hash computation.

Returns

Status of initialization

status_t DCP_HASH_Update(DCP_Type *base, dcp_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 functions blocks. If it returns kStatus_Success, the running hash has been updated (DCP has processed the input data), so the memory at the input pointer can be released back to system. The DCP context buffer is updated with the running hash and with all necessary information to support possible context switch.

Parameters

- base – DCP 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 DCP_HASH_Finish(DCP_Type *base, dcp_hash_ctx_t *ctx, uint8_t *output, size_t
                        *outputSize)
```

Finalize hashing.

Outputs the final hash (computed by DCP_HASH_Update()) and erases the context.

Parameters

- base – DCP 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

```
status_t DCP_HASH(DCP_Type *base, dcp_handle_t *handle, dcp_hash_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 or CRC32 in one function call. The function is blocking.

Parameters

- base – DCP peripheral base address
- handle – Handle used for the request.
- 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.

DCP_HASH_CAVP_COMPATIBLE

DCP_SHA_BLOCK_SIZE

DCP HASH Context size.

internal buffer block size

DCP_HASH_BLOCK_SIZE

DCP hash block size

DCP_HASH_CTX_SIZE

DCP HASH Context size.

```
struct __dcp_hash_ctx_t
```

#include <fsl_dcp.h> Storage type used to save hash context.

2.11 DCP AES non-blocking driver

```
status_t DCP_AES_EncryptEcbNonBlocking(DCP_Type *base, dcp_handle_t *handle,  
                                         dcp_work_packet_t *dcpPacket, const uint8_t  
                                         *plaintext, uint8_t *ciphertext, size_t size)
```

Encrypts AES using the ECB block mode.

Puts AES ECB encrypt work packet to DCP channel.

Parameters

- base – DCP peripheral base address
- handle – Handle used for this request.
- dcpPacket – **[out]** Memory for the DCP work packet.
- 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

kStatus_Success The work packet has been scheduled at DCP channel.

Returns

kStatus_DCP_Again The DCP channel is busy processing previous request.

```
status_t DCP_AES_DecryptEcbNonBlocking(DCP_Type *base, dcp_handle_t *handle,  
                                         dcp_work_packet_t *dcpPacket, const uint8_t  
                                         *ciphertext, uint8_t *plaintext, size_t size)
```

Decrypts AES using ECB block mode.

Puts AES ECB decrypt dcpPacket to DCP input job ring.

Parameters

- base – DCP peripheral base address
- handle – Handle used for this request.
- dcpPacket – **[out]** Memory for the DCP work packet.
- 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.

Returns

kStatus_Success The work packet has been scheduled at DCP channel.

Returns

kStatus_DCP_Again The DCP channel is busy processing previous request.

```
status_t DCP_AES_EncryptCbcNonBlocking(DCP_Type *base, dcp_handle_t *handle,  
                                         dcp_work_packet_t *dcpPacket, const uint8_t  
                                         *plaintext, uint8_t *ciphertext, size_t size, const  
                                         uint8_t *iv)
```

Encrypts AES using CBC block mode.

Puts AES CBC encrypt dcpPacket to DCP input job ring.

Parameters

- base – DCP peripheral base address
- handle – Handle used for this request. Specifies jobRing.

- dcpPacket – **[out]** Memory for the DCP work packet.
- 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

kStatus_Success The work packet has been scheduled at DCP channel.

Returns

kStatus_DCP_Again The DCP channel is busy processing previous request.

```
status_t DCP_AES_DecryptCbcNonBlocking(DCP_Type *base, dcp_handle_t *handle,  
                                         dcp_work_packet_t *dcpPacket, const uint8_t  
                                         *ciphertext, uint8_t *plaintext, size_t size, const  
                                         uint8_t *iv)
```

Decrypts AES using CBC block mode.

Puts AES CBC decrypt dcpPacket to DCP input job ring.

Parameters

- base – DCP peripheral base address
- handle – Handle used for this request. Specifies jobRing.
- dcpPacket – **[out]** Memory for the DCP work packet.
- 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

kStatus_Success The work packet has been scheduled at DCP channel.

Returns

kStatus_DCP_Again The DCP channel is busy processing previous request.

2.12 DMAMUX: Direct Memory Access Multiplexer Driver

```
void DMAMUX_Init(DMAMUX_Type *base)
```

Initializes the DMAMUX peripheral.

This function ungates the DMAMUX clock.

Parameters

- base – DMAMUX peripheral base address.

```
void DMAMUX_Deinit(DMAMUX_Type *base)
```

Deinitializes the DMAMUX peripheral.

This function gates the DMAMUX clock.

Parameters

- base – DMAMUX peripheral base address.

```
static inline void DMAMUX_EnableChannel(DMAMUX_Type *base, uint32_t channel)
```

Enables the DMAMUX channel.

This function enables the DMAMUX channel.

Parameters

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

```
static inline void DMAMUX_DisableChannel(DMAMUX_Type *base, uint32_t channel)
```

Disables the DMAMUX channel.

This function disables the DMAMUX channel.

Note: The user must disable the DMAMUX channel before configuring it.

Parameters

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

```
static inline void DMAMUX_SetSource(DMAMUX_Type *base, uint32_t channel, int32_t source)
```

Configures the DMAMUX channel source.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.
- source – Channel source, which is used to trigger the DMA transfer. User need to use the `dma_request_source_t` type as the input parameter.

```
static inline void DMAMUX_EnablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)
```

Enables the DMAMUX period trigger.

This function enables the DMAMUX period trigger feature.

Parameters

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

```
static inline void DMAMUX_DisablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)
```

Disables the DMAMUX period trigger.

This function disables the DMAMUX period trigger.

Parameters

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

```
static inline void DMAMUX_EnableAlwaysOn(DMAMUX_Type *base, uint32_t channel, bool enable)
```

Enables the DMA channel to be always ON.

This function enables the DMAMUX channel always ON feature.

Parameters

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

- `enable` – Switcher of the always ON feature. “true” means enabled, “false” means disabled.

FSL_DMAMUX_DRIVER_VERSION
DMAMUX driver version 2.1.1.

2.13 eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

`void EDMA_Init(DMA_Type *base, const edma_config_t *config)`

Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure. All emda enabled request will be cleared in this function.

Note: This function enables the minor loop map feature.

Parameters

- `base` – eDMA peripheral base address.
- `config` – A pointer to the configuration structure, see “*edma_config_t*”.

`void EDMA_Deinit(DMA_Type *base)`

Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- `base` – eDMA peripheral base address.

`void EDMA_InstallTCD(DMA_Type *base, uint32_t channel, edma_tcd_t *tcd)`

Push content of TCD structure into hardware TCD register.

Parameters

- `base` – EDMA peripheral base address.
- `channel` – EDMA channel number.
- `tcd` – Point to TCD structure.

`void EDMA_GetDefaultConfig(edma_config_t *config)`

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values.

```
config.enableContinuousLinkMode = false;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
```

Parameters

- `config` – A pointer to the eDMA configuration structure.

static inline void EDMA_EnableContinuousChannelLinkMode(DMA_Type *base, bool enable)
Enable/Disable continuous channel link mode.

Note: Do not use continuous link mode with a channel linking to itself if there is only one minor loop iteration per service request, for example, if the channel's NBYTES value is the same as either the source or destination size. The same data transfer profile can be achieved by simply increasing the NBYTES value, which provides more efficient, faster processing.

Parameters

- base – EDMA peripheral base address.
- enable – true is enable, false is disable.

static inline void EDMA_EnableMinorLoopMapping(DMA_Type *base, bool enable)
Enable/Disable minor loop mapping.

The TCDn.word2 is redefined to include individual enable fields, an offset field, and the NBYTES field.

Parameters

- base – EDMA peripheral base address.
- enable – true is enable, false is disable.

void EDMA_ResetChannel(DMA_Type *base, uint32_t channel)
Sets all TCD registers to default values.

This function sets TCD registers for this channel to default values.

Note: This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

Note: This function enables the auto stop request feature.

Parameters

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

void EDMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *edma_transfer_config_t* *config, *edma_tcd_t* *nextTcd)

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_t config;  
edma_tcd_t tcd;  
config.srcAddr = ..;  
config.destAddr = ..;  
...  
EDMA_SetTransferConfig(DMA0, channel, &config, &stcd);
```

Note: If nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the eDMA_ResetChannel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA_SetMinorOffsetConfig(DMA_Type *base, uint32_t channel, const
                               edma_minor_offset_config_t *config)
```

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – A pointer to the minor offset configuration structure.

```
void EDMA_SetChannelPreemptionConfig(DMA_Type *base, uint32_t channel, const
                                      edma_channel_preemption_config_t *config)
```

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- config – A pointer to the channel preemption configuration structure.

```
void EDMA_SetChannelLink(DMA_Type *base, uint32_t channel, edma_channel_link_type_t
                         linkType, uint32_t linkedChannel)
```

Sets the channel link for the eDMA transfer.

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- linkType – A channel link type, which can be one of the following:
 - kEDMA_LinkNone
 - kEDMA_MinorLink

- kEDMA_MajorLink
- linkedChannel – The linked channel number.

void EDMA_SetBandWidth(DMA_Type *base, uint32_t channel, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA transfer.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- bandWidth – A bandwidth setting, which can be one of the following:
 - kEDMABandwidthStallNone
 - kEDMABandwidthStall4Cycle
 - kEDMABandwidthStall8Cycle

void EDMA_SetModulo(DMA_Type *base, uint32_t channel, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA transfer.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

static inline void EDMA_EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an async request for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

static inline void EDMA_EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an auto stop request for the eDMA transfer.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).


```
void EDMA__EnableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Enables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

```
void EDMA__DisableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Disables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of the interrupt source to be set. Use the defined `edma_interrupt_enable_t` type.

```
void EDMA__SetMajorOffsetConfig(DMA_Type *base, uint32_t channel, int32_t sourceOffset,  
                                int32_t destOffset)
```

Configures the eDMA channel TCD major offset feature.

Adjustment value added to the source address at the completion of the major iteration count

Parameters

- base – eDMA peripheral base address.
- channel – edma channel number.
- sourceOffset – source address offset will be applied to source address after major loop done.
- destOffset – destination address offset will be applied to source address after major loop done.

```
void EDMA__TcdReset(edma_tcd_t *tcd)
```

Sets all fields to default values for the TCD structure.

This function sets all fields for this TCD structure to default value.

Note: This function enables the auto stop request feature.

Parameters

- tcd – Pointer to the TCD structure.

```
void EDMA__TcdSetTransferConfig(edma_tcd_t *tcd, const edma_transfer_config_t *config,  
                                edma_tcd_t *nextTcd)
```

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The TCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_t config = {  
    ...  
}  
edma_tcd_t tcd __aligned(32);  
edma_tcd_t nextTcd __aligned(32);  
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

Parameters

- tcd – Pointer to the TCD structure.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA_TcdSetMinorOffsetConfig(edma_tcd_t *tcd, const edma_minor_offset_config_t  
                                *config)
```

Configures the eDMA TCD minor offset feature.

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

- tcd – A point to the TCD structure.
- config – A pointer to the minor offset configuration structure.

```
void EDMA_TcdSetChannelLink(edma_tcd_t *tcd, edma_channel_link_type_t linkType, uint32_t  
                           linkedChannel)
```

Sets the channel link for the eDMA TCD.

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- tcd – Point to the TCD structure.
- linkType – Channel link type, it can be one of:
 - kEDMA_LinkNone
 - kEDMA_MinorLink
 - kEDMA_MajorLink
- linkedChannel – The linked channel number.

```
static inline void EDMA_TcdSetBandWidth(edma_tcd_t *tcd, edma_bandwidth_t bandWidth)
```

Sets the bandwidth for the eDMA TCD.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- *tcd* – A pointer to the TCD structure.
- *bandWidth* – A bandwidth setting, which can be one of the following:
 - *kEDMABandwidthStallNone*
 - *kEDMABandwidthStall4Cycle*
 - *kEDMABandwidthStall8Cycle*

```
void EDMA_TcdSetModulo(edma_tcd_t *tcd, edma_modulo_t srcModulo, edma_modulo_t destModulo)
```

Sets the source modulo and the destination modulo for the eDMA TCD.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- *tcd* – A pointer to the TCD structure.
- *srcModulo* – A source modulo value.
- *destModulo* – A destination modulo value.

```
static inline void EDMA_TcdEnableAutoStopRequest(edma_tcd_t *tcd, bool enable)
```

Sets the auto stop request for the eDMA TCD.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- *tcd* – A pointer to the TCD structure.
- *enable* – The command to enable (true) or disable (false).

```
void EDMA_TcdEnableInterrupts(edma_tcd_t *tcd, uint32_t mask)
```

Enables the interrupt source for the eDMA TCD.

Parameters

- *tcd* – Point to the TCD structure.
- *mask* – The mask of interrupt source to be set. Users need to use the defined *edma_interrupt_enable_t* type.

```
void EDMA_TcdDisableInterrupts(edma_tcd_t *tcd, uint32_t mask)
```

Disables the interrupt source for the eDMA TCD.

Parameters

- *tcd* – Point to the TCD structure.
- *mask* – The mask of interrupt source to be set. Users need to use the defined *edma_interrupt_enable_t* type.

void EDMA__TcdSetMajorOffsetConfig(*edma_tcd_t* *tcd, int32_t sourceOffset, int32_t destOffset)

Configures the eDMA TCD major offset feature.

Adjustment value added to the source address at the completion of the major iteration count

Parameters

- tcd – A point to the TCD structure.
- sourceOffset – source address offset will be applied to source address after major loop done.
- destOffset – destination address offset will be applied to source address after major loop done.

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

Enables the eDMA hardware channel request.

This function enables the hardware channel request.

Parameters

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

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

Disables the eDMA hardware channel request.

This function disables the hardware channel request.

Parameters

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

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

Starts the eDMA transfer by using the software trigger.

This function starts a minor loop transfer.

Parameters

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

uint32_t EDMA__GetRemainingMajorLoopCount(DMA_Type *base, uint32_t channel)

Gets the remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.

- a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTES(initially configured)
-

Parameters

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

Returns

Major loop count which has not been transferred yet for the current TCD.

```
static inline uint32_t EDMA_GetErrorStatusFlags(DMA_Type *base)
```

Gets the eDMA channel error status flags.

Parameters

- base – eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the `_edma_error_status_flags` type to decode the return variables.

```
uint32_t EDMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)
```

Gets the eDMA channel status flags.

Parameters

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

Returns

The mask of channel status flags. Users need to use the `_edma_channel_status_flags` type to decode the return variables.

```
void EDMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Clears the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of channel status to be cleared. Users need to use the defined `_edma_channel_status_flags` type.

```
void EDMA_CreateHandle(edma_handle_t *handle, DMA_Type *base, uint32_t channel)
```

Creates the eDMA handle.

This function is called if using the transactional API for eDMA. This function initializes the internal state of the eDMA handle.

Parameters

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

```
void EDMA_InstallTCDMemory(edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
```

Installs the TCDs memory pool into the eDMA handle.

This function is called after the `EDMA_CreateHandle` to use scatter/gather feature. This function shall only be used while users need to use scatter gather mode. Scatter gather mode enables EDMA to load a new transfer control block (tcd) in hardware, and automatically reconfigure that DMA channel for a new transfer. Users need to prepare tcd memory and also configure tcds using interface `EDMA_SubmitTransfer`.

Parameters

- handle – eDMA handle pointer.
- tcdPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- tcdSize – The number of TCD slots.

void EDMA_SetCallback(*edma_handle_t* *handle, *edma_callback* callback, void *userData)

Installs a callback function for the eDMA transfer.

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes. This function will be called every time one tcd finished transfer.

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

void EDMA_PrepareTransferConfig(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type *edma_transfer_t*.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- srcOffset – source address offset.
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- destOffset – destination address offset.
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.

void EDMA_PrepareTransfer(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, *edma_transfer_type_t* transferType)

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type `edma_transfer_t`.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.
- transferType – eDMA transfer type.

`status_t` EDMA_SubmitTransfer(*edma_handle_t* *handle, const *edma_transfer_config_t* *config)

Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. In scatter gather mode, call this function will add a configured tcd to the circular list of tcd pool. The tcd pools is setup by call function EDMA_InstallTCDMemory before.

Parameters

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

Return values

- `kStatus_EDMA_Success` – It means submit transfer request succeed.
- `kStatus_EDMA_QueueFull` – It means TCD queue is full. Submit transfer request is not allowed.
- `kStatus_EDMA_Busy` – It means the given channel is busy, need to submit request later.

`void` EDMA_StartTransfer(*edma_handle_t* *handle)

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- handle – eDMA handle pointer.

`void` EDMA_StopTransfer(*edma_handle_t* *handle)

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call EDMA_StartTransfer() again to resume the transfer.

Parameters

- handle – eDMA handle pointer.

`void` EDMA_AbortTransfer(*edma_handle_t* *handle)

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- handle – DMA handle pointer.

static inline uint32_t EDMA_GetUnusedTCDNumber(*edma_handle_t* *handle)

Get unused TCD slot number.

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

- handle – DMA handle pointer.

Returns

The unused tcd slot number.

static inline uint32_t EDMA_GetNextTCDAddress(*edma_handle_t* *handle)

Get the next tcd address.

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

- handle – DMA handle pointer.

Returns

The next TCD address.

void EDMA_HandleIRQ(*edma_handle_t* *handle)

eDMA IRQ handler for the current major loop transfer completion.

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

Note: For the case using TCD queue, when the major iteration count is exhausted, additional operations are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As sga and sga_index are calculated based on the DLAST_SGA bitfield lies in the TCD_CSR register, the sga_index in this case should be 2 (DLAST_SGA of TCD[1] stores the address of TCD[2]). Thus, the “tcdUsed” updated should be (tcdUsed - 2U) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and tcdUsed updated are identical for them. tcdUsed are both 0 in this case as no TCD to be loaded.

See the “eDMA basic data flow” in the eDMA Functional description section of the Reference Manual for further details.

Parameters

- handle – eDMA handle pointer.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.4.7.

enum __edma_transfer_size

eDMA transfer configuration

Values:

enumerator kEDMA_TransferSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kEDMA_TransferSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kEDMA_TransferSize4Bytes

Source/Destination data transfer size is 4 bytes every time

enumerator kEDMA_TransferSize8Bytes

Source/Destination data transfer size is 8 bytes every time

enumerator kEDMA_TransferSize16Bytes

Source/Destination data transfer size is 16 bytes every time

enumerator kEDMA_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enum _edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes

Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes

Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes

Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes

Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes

Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes

Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes

Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes

Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes

Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes

Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes

Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes

Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes

Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes

Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes

Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes

Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes

Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes

Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes

Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes

Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes

Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes

Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes

Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes

Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes

Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes

Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes

Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes

Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes

Circular buffer size is 2 G bytes.

enum _edma_bandwidth

Bandwidth control.

Values:

enumerator kEDMA_BandwidthStallNone

No eDMA engine stalls.

enumerator kEDMA_BandwidthStall4Cycle

eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_BandwidthStall8Cycle

eDMA engine stalls for 8 cycles after each read/write.

enum _edma_channel_link_type

Channel link type.

Values:

enumerator kEDMA_LinkNone

No channel link

enumerator kEDMA_MinorLink

Channel link after each minor loop

enumerator kEDMA_MajorLink

Channel link while major loop count exhausted

_edma_channel_status_flags eDMA channel status flags.

Values:

enumerator kEDMA_DoneFlag

DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag

eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag

eDMA interrupt flag, set while an interrupt occurred of this channel

_edma_error_status_flags eDMA channel error status flags.

Values:

enumerator kEDMA_DestinationBusErrorFlag

Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag

Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag

Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag

NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag

Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag

Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag

Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag

Source address not aligned with source size

enumerator kEDMA_ErrorChannelFlag

Error channel number of the cancelled channel number

enumerator kEDMA_ChannelPriorityErrorFlag

Channel priority is not unique.

enumerator kEDMA_TransferCanceledFlag

Transfer cancelled

enumerator kEDMA_ValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

enum _edma_interrupt_enable

eDMA interrupt source

Values:

enumerator kEDMA_ErrorInterruptEnable

Enable interrupt while channel error occurs.

enumerator kEDMA_MajorInterruptEnable

Enable interrupt while major count exhausted.

enumerator kEDMA_HalfInterruptEnable

Enable interrupt while major count to half value.

enum _edma_transfer_type

eDMA transfer type

Values:

enumerator kEDMA_MemoryToMemory

Transfer from memory to memory

enumerator kEDMA_PeripheralToMemory

Transfer from peripheral to memory

enumerator kEDMA_MemoryToPeripheral

Transfer from memory to peripheral

enumerator kEDMA_PeripheralToPeripheral

Transfer from Peripheral to peripheral

_edma_transfer_status eDMA transfer status

Values:

enumerator kStatus_EDMA_QueueFull

TCD queue is full.

enumerator kStatus_EDMA_Busy

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

typedef enum _edma_transfer_size edma_transfer_size_t

eDMA transfer configuration

typedef enum _edma_modulo edma_modulo_t

eDMA modulo configuration

```
typedef enum _edma_bandwidth edma__bandwidth__t
```

Bandwidth control.

```
typedef enum _edma_channel_link_type edma__channel__link__type__t
```

Channel link type.

```
typedef enum _edma_interrupt_enable edma__interrupt__enable__t
```

eDMA interrupt source

```
typedef enum _edma_transfer_type edma__transfer__type__t
```

eDMA transfer type

```
typedef struct _edma_config edma__config__t
```

eDMA global configuration structure.

```
typedef struct _edma_transfer_config edma__transfer__config__t
```

eDMA transfer configuration

This structure configures the source/destination transfer attribute.

```
typedef struct _edma_channel_Preemption_config edma__channel__Preemption__config__t
```

eDMA channel priority configuration

```
typedef struct _edma_minor_offset_config edma__minor__offset__config__t
```

eDMA minor offset configuration

```
typedef struct _edma_tcd edma__tcd__t
```

eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

```
typedef void (*edma__callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcDs)
```

Define callback function for eDMA.

This callback function is called in the EDMA interrupt handle. In normal mode, run into callback function means the transfer users need is done. In scatter gather mode, run into callback function means a transfer control block (tcd) is finished. Not all transfer finished, users can get the finished tcd numbers using interface `EDMA_GetUnusedTCDNumber`.

Param handle

EDMA handle pointer, users shall not touch the values inside.

Param userData

The callback user parameter pointer. Users can use this parameter to involve things users need to change in EDMA callback function.

Param transferDone

If the current loaded transfer done. In normal mode it means if all transfer done. In scatter gather mode, this parameter shows is the current transfer block in EDMA register is done. As the load of core is different, it will be different if the new tcd loaded into EDMA registers while this callback called. If true, it always means new tcd still not loaded into registers, while false means new tcd already loaded into registers.

Param tcDs

How many tcDs are done from the last callback. This parameter only used in scatter gather mode. It tells user how many tcDs are finished between the last callback and this.

```
typedef struct _edma_handle edma__handle__t
```

eDMA transfer handle structure

DMA_DCHPRI_INDEX(channel)

Compute the offset unit from DCHPRI3.

struct __edma_config

#include <fsl_edma.h> eDMA global configuration structure.

Public Members

bool enableContinuousLinkMode

Enable (true) continuous link mode. Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.

bool enableHaltOnError

Enable (true) transfer halt on error. Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

bool enableRoundRobinArbitration

Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection

bool enableDebugMode

Enable(true) eDMA debug mode. When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

struct __edma_transfer_config

#include <fsl_edma.h> eDMA transfer configuration

This structure configures the source/destination transfer attribute.

Public Members

uint32_t srcAddr

Source data address.

uint32_t destAddr

Destination data address.

edma_transfer_size_t srcTransferSize

Source data transfer size.

edma_transfer_size_t destTransferSize

Destination data transfer size.

int16_t srcOffset

Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.

int16_t destOffset

Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.

uint32_t minorLoopBytes

Bytes to transfer in a minor loop

uint32_t majorLoopCounts

Major loop iteration count.

struct __edma_channel_Preemption_config

#include <fsl_edma.h> eDMA channel priority configuration

Public Members

bool enableChannelPreemption

If true: a channel can be suspended by other channel with higher priority

bool enablePreemptAbility

If true: a channel can suspend other channel with low priority

uint8_t channelPriority

Channel priority

struct __edma__minor_offset_config

#include <fsl_edma.h> eDMA minor offset configuration

Public Members

bool enableSrcMinorOffset

Enable(true) or Disable(false) source minor loop offset.

bool enableDestMinorOffset

Enable(true) or Disable(false) destination minor loop offset.

uint32_t minorOffset

Offset for a minor loop mapping.

struct __edma_tcd

#include <fsl_edma.h> eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

Public Members

__IO uint32_t SADDR

SADDR register, used to save source address

__IO uint16_t SOFF

SOFF register, save offset bytes every transfer

__IO uint16_t ATTR

ATTR register, source/destination transfer size and modulo

__IO uint32_t NBYTES

Nbytes register, minor loop length in bytes

__IO uint32_t SLAST

SLAST register

__IO uint32_t DADDR

DADDR register, used for destination address

__IO uint16_t DOFF

DOFF register, used for destination offset

__IO uint16_t CITER

CITER register, current minor loop numbers, for unfinished minor loop.

__IO uint32_t DLAST_SGA

DLASTSGA register, next tcd address used in scatter-gather mode

__IO uint16_t CSR
CSR register, for TCD control status

__IO uint16_t BITER
BITER register, begin minor loop count.

struct _edma_handle
#include <fsl_edma.h> eDMA transfer handle structure

Public Members

edma_callback callback
Callback function for major count exhausted.

void *userData
Callback function parameter.

DMA_Type *base
eDMA peripheral base address.

edma_tcd_t *tcdPool
Pointer to memory stored TCDs.

uint8_t channel
eDMA channel number.

volatile int8_t header
The first TCD index. Should point to the next TCD to be loaded into the eDMA engine.

volatile int8_t tail
The last TCD index. Should point to the next TCD to be stored into the memory pool.

volatile int8_t tcdUsed
The number of used TCD slots. Should reflect the number of TCDs can be used/loaded in the memory.

volatile int8_t tcdSize
The total number of TCD slots in the queue.

uint8_t flags
The status of the current channel.

2.14 EWM: External Watchdog Monitor Driver

void EWM_Init(EWM_Type *base, const *ewm_config_t* *config)
Initializes the EWM peripheral.

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
ewm_config_t config;  
EWM_GetDefaultConfig(&config);  
config.compareHighValue = 0xAAU;  
EWM_Init(ewm_base,&config);
```


Parameters

- base – EWM peripheral base address
- config – The configuration of the EWM

void EWM_Deinit(EWM_Type *base)

Deinitializes the EWM peripheral.

This function is used to shut down the EWM.

Parameters

- base – EWM peripheral base address

void EWM_GetDefaultConfig(*ewm_config_t* *config)

Initializes the EWM configuration structure.

This function initializes the EWM configuration structure to default values. The default values are as follows.

```
ewmConfig->enableEwm = true;
ewmConfig->enableEwmInput = false;
ewmConfig->setInputAssertLogic = false;
ewmConfig->enableInterrupt = false;
ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
ewmConfig->prescaler = 0;
ewmConfig->compareLowValue = 0;
ewmConfig->compareHighValue = 0xFEU;
```

See also:

[ewm_config_t](#)

Parameters

- config – Pointer to the EWM configuration structure.

static inline void EWM_EnableInterrupts(EWM_Type *base, uint32_t mask)

Enables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to enable The parameter can be combination of the following source if defined
 - kEWM_InterruptEnable

static inline void EWM_DisableInterrupts(EWM_Type *base, uint32_t mask)

Disables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to disable The parameter can be combination of the following source if defined
 - kEWM_InterruptEnable

static inline uint32_t EWM_GetStatusFlags(EWM_Type *base)

Gets all status flags.

This function gets all status flags.

This is an example for getting the running flag.

```
uint32_t status;  
status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
```

See also:

`_ewm_status_flags_t`

- True: a related status flag has been set.
- False: a related status flag is not set.

Parameters

- base – EWM peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

void EWM_Refresh(EWM_Type *base)

Services the EWM.

This function resets the EWM counter to zero.

Parameters

- base – EWM peripheral base address

FSL_EWM_DRIVER_VERSION

EWM driver version 2.0.4.

enum _ewm_lpo_clock_source

Describes EWM clock source.

Values:

enumerator kEWM_LpoClockSource0
EWM clock sourced from lpo_clk[0]

enumerator kEWM_LpoClockSource1
EWM clock sourced from lpo_clk[1]

enumerator kEWM_LpoClockSource2
EWM clock sourced from lpo_clk[2]

enumerator kEWM_LpoClockSource3
EWM clock sourced from lpo_clk[3]

enum _ewm_interrupt_enable_t

EWM interrupt configuration structure with default settings all disabled.

This structure contains the settings for all of EWM interrupt configurations.

Values:

enumerator kEWM_InterruptEnable
Enable the EWM to generate an interrupt

enum `_ewm_status_flags_t`

EWM status flags.

This structure contains the constants for the EWM status flags for use in the EWM functions.

Values:

enumerator `kEWM_RunningFlag`

Running flag, set when EWM is enabled

typedef enum `_ewm_lpo_clock_source` `ewm_lpo_clock_source_t`

Describes EWM clock source.

typedef struct `_ewm_config` `ewm_config_t`

Data structure for EWM configuration.

This structure is used to configure the EWM.

struct `_ewm_config`

#include <fsl_ewm.h> Data structure for EWM configuration.

This structure is used to configure the EWM.

Public Members

bool `enableEwm`

Enable EWM module

bool `enableEwmInput`

Enable EWM_in input

bool `setInputAssertLogic`

EWM_in signal assertion state

bool `enableInterrupt`

Enable EWM interrupt

`ewm_lpo_clock_source_t` `clockSource`

Clock source select

uint8_t `prescaler`

Clock prescaler value

uint8_t `compareLowValue`

Compare low-register value

uint8_t `compareHighValue`

Compare high-register value

2.15 FlexIO: FlexIO Driver

2.16 FlexIO Driver

void `FLEXIO_GetDefaultConfig(flexio_config_t *userConfig)`

Gets the default configuration to configure the FlexIO module. The configuration can be used directly to call the `FLEXIO_Configure()`.

Example:

```
flexio_config_t config;  
FLEXIO_GetDefaultConfig(&config);
```

Parameters

- userConfig – pointer to flexio_config_t structure

void FLEXIO_Init(FLEXIO_Type *base, const flexio_config_t *userConfig)

Configures the FlexIO with a FlexIO configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_GetDefaultConfig().

Example

```
flexio_config_t config = {  
.enableFlexio = true,  
.enableInDoze = false,  
.enableInDebug = true,  
.enableFastAccess = false  
};  
FLEXIO_Configure(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- userConfig – pointer to flexio_config_t structure

void FLEXIO_Deinit(FLEXIO_Type *base)

Gates the FlexIO clock. Call this API to stop the FlexIO clock.

Note: After calling this API, call the FLEXIO_Init to use the FlexIO module.

Parameters

- base – FlexIO peripheral base address

uint32_t FLEXIO_GetInstance(FLEXIO_Type *base)

Get instance number for FLEXIO module.

Parameters

- base – FLEXIO peripheral base address.

void FLEXIO_Reset(FLEXIO_Type *base)

Resets the FlexIO module.

Parameters

- base – FlexIO peripheral base address

static inline void FLEXIO_Enable(FLEXIO_Type *base, bool enable)

Enables the FlexIO module operation.

Parameters

- base – FlexIO peripheral base address
- enable – true to enable, false to disable.

static inline uint32_t FLEXIO_ReadPinInput(FLEXIO_Type *base)

Reads the input data on each of the FlexIO pins.

Parameters

- base – FlexIO peripheral base address

Returns

FlexIO pin input data

```
static inline uint8_t FLEXIO_GetShifterState(FLEXIO_Type *base)
```

Gets the current state pointer for state mode use.

Parameters

- base – FlexIO peripheral base address

Returns

current State pointer

```
void FLEXIO_SetShifterConfig(FLEXIO_Type *base, uint8_t index, const flexio_shifter_config_t *shifterConfig)
```

Configures the shifter with the shifter configuration. The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {
    .timerSelect = 0,
    .timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,
    .pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
    .pinPolarity = kFLEXIO_PinActiveLow,
    .shifterMode = kFLEXIO_ShifterModeTransmit,
    .inputSource = kFLEXIO_ShifterInputFromPin,
    .shifterStop = kFLEXIO_ShifterStopBitHigh,
    .shifterStart = kFLEXIO_ShifterStartBitLow
};
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Shifter index
- shifterConfig – Pointer to flexio_shifter_config_t structure

```
void FLEXIO_SetTimerConfig(FLEXIO_Type *base, uint8_t index, const flexio_timer_config_t *timerConfig)
```

Configures the timer with the timer configuration. The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

Example

```
flexio_timer_config_t config = {
    .triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(0),
    .triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,
    .triggerSource = kFLEXIO_TimerTriggerSourceInternal,
    .pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
    .pinSelect = 0,
    .pinPolarity = kFLEXIO_PinActiveHigh,
    .timerMode = kFLEXIO_TimerModeDual8BitBaudBit,
    .timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,
    .timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput,
    .timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,
    .timerDisable = kFLEXIO_TimerDisableOnTimerCompare,
    .timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,
    .timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,
```

(continues on next page)

(continued from previous page)

```
.timerStart = kFLEXIO_TimerStartBitEnabled
};
FLEXIO_SetTimerConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Timer index
- timerConfig – Pointer to the flexio_timer_config_t structure

```
static inline void FLEXIO_SetClockMode(FLEXIO_Type *base, uint8_t index,
                                       flexio_timer_decrement_source_t clocksource)
```

This function set the value of the prescaler on flexio channels.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- index – Timer index
- clocksource – Set clock value

```
static inline void FLEXIO_EnableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

```
static inline void FLEXIO_DisableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Disables the shifter status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

```
static inline void FLEXIO_EnableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter error interrupt. The interrupt generates when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_DisableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the shifter error interrupt. The interrupt won't generate when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_EnableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Enables the timer status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_DisableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the timer status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline uint32_t FLEXIO_GetShifterStatusFlags(FLEXIO_Type *base)
Gets the shifter status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter status flags

static inline void FLEXIO_ClearShifterStatusFlags(FLEXIO_Type *base, uint32_t mask)
Clears the shifter status flags.

Note: For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetShifterErrorFlags(FLEXIO_Type *base)

Gets the shifter error flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter error flags

static inline void FLEXIO_ClearShifterErrorFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter error flags.

Note: For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetTimerStatusFlags(FLEXIO_Type *base)

Gets the timer status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Timer status flags

static inline void FLEXIO_ClearTimerStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the timer status flags.

Note: For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_EnableShifterStatusDMA(FLEXIO_Type *base, uint32_t mask, bool enable)

Enables/disables the shifter status DMA. The DMA request generates when the corresponding SSF is set.

Note: For multiple shifter status DMA enables, for example, calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$
- enable – True to enable, false to disable.

uint32_t FLEXIO_GetShifterBufferAddress(FLEXIO_Type *base, flexio_shifter_buffer_type_t type, uint8_t index)

Gets the shifter buffer address for the DMA transfer usage.

Parameters

- base – FlexIO peripheral base address
- type – Shifter type of flexio_shifter_buffer_type_t
- index – Shifter index

Returns

Corresponding shifter buffer index

status_t FLEXIO_RegisterHandleIRQ(void *base, void *handle, flexio_isr_t isr)

Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- handle – Pointer to the handler for FlexIO simulated peripheral.
- isr – FlexIO simulated peripheral interrupt handler.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_UnregisterHandleIRQ(void *base)

Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

static inline void FLEXIO_ClearPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 0.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_SetPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 1.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_TogglePortOutput(FLEXIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple FLEXIO pins.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_PinWrite(FLEXIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the FLEXIO pins to the logic 1 or 0.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- output – FLEXIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void FLEXIO_EnablePinOutput(FLEXIO_Type *base, uint32_t pin)

Enables the FLEXIO output pin function.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

static inline uint32_t FLEXIO_PinRead(FLEXIO_Type *base, uint32_t pin)

Reads the current input value of the FLEXIO pin.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline uint32_t FLEXIO_GetPinStatus(FLEXIO_Type *base, uint32_t pin)

Gets the FLEXIO input pin status.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input status

- 0: corresponding pin input capture no status.
- 1: corresponding pin input capture rising or falling edge.

static inline void FLEXIO_SetPinLevel(FLEXIO_Type *base, uint8_t pin, bool level)

Sets the FLEXIO output pin level.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.
- level – FlexIO output pin level to set, can be either 0 or 1.

static inline bool FLEXIO_GetPinOverride(const FLEXIO_Type *const base, uint8_t pin)

Gets the enabled status of a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.

Return values

FlexIO – port enabled status

- 0: corresponding output pin is in disabled state.
- 1: corresponding output pin is in enabled state.

static inline void FLEXIO_ConfigPinOverride(FLEXIO_Type *base, uint8_t pin, bool enabled)

Enables or disables a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – Flexio pin number.
- enabled – Enable or disable the FlexIO pin.

static inline void FLEXIO_ClearPortStatus(FLEXIO_Type *base, uint32_t mask)

Clears the multiple FLEXIO input pins status.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

FSL_FLEXIO_DRIVER_VERSION

FlexIO driver version.

enum _flexio_timer_trigger_polarity

Define time of timer trigger polarity.

Values:

enumerator kFLEXIO_TimerTriggerPolarityActiveHigh
Active high.

enumerator kFLEXIO_TimerTriggerPolarityActiveLow
Active low.

enum _flexio_timer_trigger_source

Define type of timer trigger source.

Values:

enumerator kFLEXIO_TimerTriggerSourceExternal
External trigger selected.

enumerator kFLEXIO__TimerTriggerSourceInternal
Internal trigger selected.

enum _flexio_pin_config
Define type of timer/shifter pin configuration.

Values:

enumerator kFLEXIO__PinConfigOutputDisabled
Pin output disabled.

enumerator kFLEXIO__PinConfigOpenDrainOrBidirection
Pin open drain or bidirectional output enable.

enumerator kFLEXIO__PinConfigBidirectionOutputData
Pin bidirectional output data.

enumerator kFLEXIO__PinConfigOutput
Pin output.

enum _flexio_pin_polarity
Definition of pin polarity.

Values:

enumerator kFLEXIO__PinActiveHigh
Active high.

enumerator kFLEXIO__PinActiveLow
Active low.

enum _flexio_timer_mode
Define type of timer work mode.

Values:

enumerator kFLEXIO__TimerModeDisabled
Timer Disabled.

enumerator kFLEXIO__TimerModeDual8BitBaudBit
Dual 8-bit counters baud/bit mode.

enumerator kFLEXIO__TimerModeDual8BitPWM
Dual 8-bit counters PWM mode.

enumerator kFLEXIO__TimerModeSingle16Bit
Single 16-bit counter mode.

enumerator kFLEXIO__TimerModeDual8BitPWMLow
Dual 8-bit counters PWM Low mode.

enum _flexio_timer_output
Define type of timer initial output or timer reset condition.

Values:

enumerator kFLEXIO__TimerOutputOneNotAffectedByReset
Logic one when enabled and is not affected by timer reset.

enumerator kFLEXIO__TimerOutputZeroNotAffectedByReset
Logic zero when enabled and is not affected by timer reset.

enumerator kFLEXIO__TimerOutputOneAffectedByReset
Logic one when enabled and on timer reset.

enumerator kFLEXIO__TimerOutputZeroAffectedByReset

Logic zero when enabled and on timer reset.

enum __flexio_timer_decrement_source

Define type of timer decrement.

Values:

enumerator kFLEXIO__TimerDecSrcOnFlexIOClockShiftTimerOutput

Decrement counter on FlexIO clock, Shift clock equals Timer output.

enumerator kFLEXIO__TimerDecSrcOnTriggerInputShiftTimerOutput

Decrement counter on Trigger input (both edges), Shift clock equals Timer output.

enumerator kFLEXIO__TimerDecSrcOnPinInputShiftPinInput

Decrement counter on Pin input (both edges), Shift clock equals Pin input.

enumerator kFLEXIO__TimerDecSrcOnTriggerInputShiftTriggerInput

Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

enum __flexio_timer_reset_condition

Define type of timer reset condition.

Values:

enumerator kFLEXIO__TimerResetNever

Timer never reset.

enumerator kFLEXIO__TimerResetOnTimerPinEqualToTimerOutput

Timer reset on Timer Pin equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerTriggerEqualToTimerOutput

Timer reset on Timer Trigger equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerPinRisingEdge

Timer reset on Timer Pin rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerRisingEdge

Timer reset on Trigger rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerBothEdge

Timer reset on Trigger rising or falling edge.

enum __flexio_timer_disable_condition

Define type of timer disable condition.

Values:

enumerator kFLEXIO__TimerDisableNever

Timer never disabled.

enumerator kFLEXIO__TimerDisableOnPreTimerDisable

Timer disabled on Timer N-1 disable.

enumerator kFLEXIO__TimerDisableOnTimerCompare

Timer disabled on Timer compare.

enumerator kFLEXIO__TimerDisableOnTimerCompareTriggerLow

Timer disabled on Timer compare and Trigger Low.

enumerator kFLEXIO__TimerDisableOnPinBothEdge

Timer disabled on Pin rising or falling edge.

enumerator kFLEXIO__TimerDisableOnPinBothEdgeTriggerHigh

Timer disabled on Pin rising or falling edge provided Trigger is high.

enumerator kFLEXIO__TimerDisableOnTriggerFallingEdge

Timer disabled on Trigger falling edge.

enum _flexio_timer_enable_condition

Define type of timer enable condition.

Values:

enumerator kFLEXIO__TimerEnabledAlways

Timer always enabled.

enumerator kFLEXIO__TimerEnableOnPrevTimerEnable

Timer enabled on Timer N-1 enable.

enumerator kFLEXIO__TimerEnableOnTriggerHigh

Timer enabled on Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerHighPinHigh

Timer enabled on Trigger high and Pin high.

enumerator kFLEXIO__TimerEnableOnPinRisingEdge

Timer enabled on Pin rising edge.

enumerator kFLEXIO__TimerEnableOnPinRisingEdgeTriggerHigh

Timer enabled on Pin rising edge and Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerRisingEdge

Timer enabled on Trigger rising edge.

enumerator kFLEXIO__TimerEnableOnTriggerBothEdge

Timer enabled on Trigger rising or falling edge.

enum _flexio_timer_stop_bit_condition

Define type of timer stop bit generate condition.

Values:

enumerator kFLEXIO__TimerStopBitDisabled

Stop bit disabled.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompare

Stop bit is enabled on timer compare.

enumerator kFLEXIO__TimerStopBitEnableOnTimerDisable

Stop bit is enabled on timer disable.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompareDisable

Stop bit is enabled on timer compare and timer disable.

enum _flexio_timer_start_bit_condition

Define type of timer start bit generate condition.

Values:

enumerator kFLEXIO__TimerStartBitDisabled

Start bit disabled.

enumerator kFLEXIO__TimerStartBitEnabled

Start bit enabled.

enum _flexio_timer_output_state

FlexIO as PWM channel output state.

Values:

enumerator kFLEXIO_PwmLow

The output state of PWM channel is low

enumerator kFLEXIO_PwmHigh

The output state of PWM channel is high

enum _flexio_shifter_timer_polarity

Define type of timer polarity for shifter control.

Values:

enumerator kFLEXIO_ShifterTimerPolarityOnPositive

Shift on positive edge of shift clock.

enumerator kFLEXIO_ShifterTimerPolarityOnNegative

Shift on negative edge of shift clock.

enum _flexio_shifter_mode

Define type of shifter working mode.

Values:

enumerator kFLEXIO_ShifterDisabled

Shifter is disabled.

enumerator kFLEXIO_ShifterModeReceive

Receive mode.

enumerator kFLEXIO_ShifterModeTransmit

Transmit mode.

enumerator kFLEXIO_ShifterModeMatchStore

Match store mode.

enumerator kFLEXIO_ShifterModeMatchContinuous

Match continuous mode.

enumerator kFLEXIO_ShifterModeState

SHIFTBUF contents are used for storing programmable state attributes.

enumerator kFLEXIO_ShifterModeLogic

SHIFTBUF contents are used for implementing programmable logic look up table.

enum _flexio_shifter_input_source

Define type of shifter input source.

Values:

enumerator kFLEXIO_ShifterInputFromPin

Shifter input from pin.

enumerator kFLEXIO_ShifterInputFromNextShifterOutput

Shifter input from Shifter N+1.

enum _flexio_shifter_stop_bit

Define of STOP bit configuration.

Values:

enumerator kFLEXIO__ShifterStopBitDisable

Disable shifter stop bit.

enumerator kFLEXIO__ShifterStopBitLow

Set shifter stop bit to logic low level.

enumerator kFLEXIO__ShifterStopBitHigh

Set shifter stop bit to logic high level.

enum _flexio__shifter__start__bit

Define type of START bit configuration.

Values:

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnEnable

Disable shifter start bit, transmitter loads data on enable.

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnShift

Disable shifter start bit, transmitter loads data on first shift.

enumerator kFLEXIO__ShifterStartBitLow

Set shifter start bit to logic low level.

enumerator kFLEXIO__ShifterStartBitHigh

Set shifter start bit to logic high level.

enum _flexio__shifter__buffer__type

Define FlexIO shifter buffer type.

Values:

enumerator kFLEXIO__ShifterBuffer

Shifter Buffer N Register.

enumerator kFLEXIO__ShifterBufferBitSwapped

Shifter Buffer N Bit Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferByteSwapped

Shifter Buffer N Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferBitByteSwapped

Shifter Buffer N Bit Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleByteSwapped

Shifter Buffer N Nibble Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferHalfWordSwapped

Shifter Buffer N Half Word Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleSwapped

Shifter Buffer N Nibble Swapped Register.

enum _flexio__gpio__direction

FLEXIO gpio direction definition.

Values:

enumerator kFLEXIO__DigitalInput

Set current pin as digital input

enumerator kFLEXIO__DigitalOutput

Set current pin as digital output

enum *flexio_pin_input_config*

FLEXIO gpio input config.

Values:

enumerator *kFLEXIO__InputInterruptDisabled*

Interrupt request is disabled.

enumerator *kFLEXIO__InputInterruptEnable*

Interrupt request is enable.

enumerator *kFLEXIO__FlagRisingEdgeEnable*

Input pin flag on rising edge.

enumerator *kFLEXIO__FlagFallingEdgeEnable*

Input pin flag on falling edge.

typedef enum *flexio_timer_trigger_polarity* *flexio_timer_trigger_polarity_t*

Define time of timer trigger polarity.

typedef enum *flexio_timer_trigger_source* *flexio_timer_trigger_source_t*

Define type of timer trigger source.

typedef enum *flexio_pin_config* *flexio_pin_config_t*

Define type of timer/shifter pin configuration.

typedef enum *flexio_pin_polarity* *flexio_pin_polarity_t*

Definition of pin polarity.

typedef enum *flexio_timer_mode* *flexio_timer_mode_t*

Define type of timer work mode.

typedef enum *flexio_timer_output* *flexio_timer_output_t*

Define type of timer initial output or timer reset condition.

typedef enum *flexio_timer_decrement_source* *flexio_timer_decrement_source_t*

Define type of timer decrement.

typedef enum *flexio_timer_reset_condition* *flexio_timer_reset_condition_t*

Define type of timer reset condition.

typedef enum *flexio_timer_disable_condition* *flexio_timer_disable_condition_t*

Define type of timer disable condition.

typedef enum *flexio_timer_enable_condition* *flexio_timer_enable_condition_t*

Define type of timer enable condition.

typedef enum *flexio_timer_stop_bit_condition* *flexio_timer_stop_bit_condition_t*

Define type of timer stop bit generate condition.

typedef enum *flexio_timer_start_bit_condition* *flexio_timer_start_bit_condition_t*

Define type of timer start bit generate condition.

typedef enum *flexio_timer_output_state* *flexio_timer_output_state_t*

FlexIO as PWM channel output state.

typedef enum *flexio_shifter_timer_polarity* *flexio_shifter_timer_polarity_t*

Define type of timer polarity for shifter control.

typedef enum *flexio_shifter_mode* *flexio_shifter_mode_t*

Define type of shifter working mode.

```
typedef enum _flexio_shifter_input_source flexio_shifter_input_source_t
```

Define type of shifter input source.

```
typedef enum _flexio_shifter_stop_bit flexio_shifter_stop_bit_t
```

Define of STOP bit configuration.

```
typedef enum _flexio_shifter_start_bit flexio_shifter_start_bit_t
```

Define type of START bit configuration.

```
typedef enum _flexio_shifter_buffer_type flexio_shifter_buffer_type_t
```

Define FlexIO shifter buffer type.

```
typedef struct _flexio_config flexio_config_t
```

Define FlexIO user configuration structure.

```
typedef struct _flexio_timer_config flexio_timer_config_t
```

Define FlexIO timer configuration structure.

```
typedef struct _flexio_shifter_config flexio_shifter_config_t
```

Define FlexIO shifter configuration structure.

```
typedef enum _flexio_gpio_direction flexio_gpio_direction_t
```

FLEXIO gpio direction definition.

```
typedef enum _flexio_pin_input_config flexio_pin_input_config_t
```

FLEXIO gpio input config.

```
typedef struct _flexio_gpio_config flexio_gpio_config_t
```

The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use `inputConfig` param. If configured as an output pin, use `outputLogic`.

```
typedef void (*flexio_isr_t)(void *base, void *handle)
```

typedef for FlexIO simulated driver interrupt handler.

```
FLEXIO_Type *const s_flexioBases[]
```

Pointers to flexio bases for each instance.

```
const clock_ip_name_t s_flexioClocks[]
```

Pointers to flexio clocks for each instance.

```
void FLEXIO_SetPinConfig(FLEXIO_Type *base, uint32_t pin, flexio_gpio_config_t *config)
```

Configure a FLEXIO pin used by the board.

To Config the FLEXIO PIN, define a pin configuration, as either input or output, in the user file. Then, call the `FLEXIO_SetPinConfig()` function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalInput,
    0U,
    kFLEXIO_FlagRisingEdgeEnable | kFLEXIO_InputInterruptEnable,
}
Define a digital output pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalOutput,
    0U,
```

(continues on next page)

(continued from previous page)

```
0U
}
```

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- config – FLEXIO pin configuration pointer.

FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x)

Calculate FlexIO timer trigger.

FLEXIO_TIMER_TRIGGER_SEL_SHIFThSTAT(x)

FLEXIO_TIMER_TRIGGER_SEL_TIMn(x)

struct _flexio_config_

#include <fsl_flexio.h> Define FlexIO user configuration structure.

Public Members

bool enableFlexio

Enable/disable FlexIO module

bool enableInDoze

Enable/disable FlexIO operation in doze mode

bool enableInDebug

Enable/disable FlexIO operation in debug mode

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

struct _flexio_timer_config

#include <fsl_flexio.h> Define FlexIO timer configuration structure.

Public Members

uint32_t triggerSelect

The internal trigger selection number using MACROs.

flexio_timer_trigger_polarity_t triggerPolarity

Trigger Polarity.

flexio_timer_trigger_source_t triggerSource

Trigger Source, internal (see 'trgsel') or external.

flexio_pin_config_t pinConfig

Timer Pin Configuration.

uint32_t pinSelect

Timer Pin number Select.

flexio_pin_polarity_t pinPolarity

Timer Pin Polarity.

flexio_timer_mode_t timerMode

Timer work Mode.

flexio_timer_output_t timerOutput

Configures the initial state of the Timer Output and whether it is affected by the Timer reset.

flexio_timer_decrement_source_t timerDecrement

Configures the source of the Timer decrement and the source of the Shift clock.

flexio_timer_reset_condition_t timerReset

Configures the condition that causes the timer counter (and optionally the timer output) to be reset.

flexio_timer_disable_condition_t timerDisable

Configures the condition that causes the Timer to be disabled and stop decrementing.

flexio_timer_enable_condition_t timerEnable

Configures the condition that causes the Timer to be enabled and start decrementing.

flexio_timer_stop_bit_condition_t timerStop

Timer STOP Bit generation.

flexio_timer_start_bit_condition_t timerStart

Timer STRAT Bit generation.

uint32_t timerCompare

Value for Timer Compare N Register.

struct *_flexio_shifter_config*

#include <fsl_flexio.h> Define FlexIO shifter configuration structure.

Public Members

uint32_t timerSelect

Selects which Timer is used for controlling the logic/shift register and generating the Shift clock.

flexio_shifter_timer_polarity_t timerPolarity

Timer Polarity.

flexio_pin_config_t pinConfig

Shifter Pin Configuration.

uint32_t pinSelect

Shifter Pin number Select.

flexio_pin_polarity_t pinPolarity

Shifter Pin Polarity.

flexio_shifter_mode_t shifterMode

Configures the mode of the Shifter.

uint32_t parallelWidth

Configures the parallel width when using parallel mode.

flexio_shifter_input_source_t inputSource

Selects the input source for the shifter.

flexio_shifter_stop_bit_t shifterStop

Shifter STOP bit.

flexio_shifter_start_bit_t shifterStart

Shifter START bit.

struct *_flexio_gpio_config*

#include <fsl_flexio.h> The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

Public Members

flexio_gpio_direction_t pinDirection

FLEXIO pin direction, input or output

uint8_t outputLogic

Set a default output logic, which has no use in input

uint8_t inputConfig

Set an input config

2.17 FlexIO eDMA I2S Driver

```
void FLEXIO_I2S_TransferTxCreateHandleEDMA(FLEXIO_I2S_Type *base,
                                           flexio_i2s_edma_handle_t *handle,
                                           flexio_i2s_edma_callback_t callback, void
                                           *userData, edma_handle_t *dmaHandle)
```

Initializes the FlexIO I2S eDMA handle.

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer.
- callback – FlexIO I2S eDMA callback function called while finished a block.
- userData – User parameter for callback.
- dmaHandle – eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

```
void FLEXIO_I2S_TransferRxCreateHandleEDMA(FLEXIO_I2S_Type *base,
                                           flexio_i2s_edma_handle_t *handle,
                                           flexio_i2s_edma_callback_t callback, void
                                           *userData, edma_handle_t *dmaHandle)
```

Initializes the FlexIO I2S Rx eDMA handle.

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer.

- callback – FlexIO I2S eDMA callback function called while finished a block.
- userData – User parameter for callback.
- dmaHandle – eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

```
void FLEXIO_I2S_TransferSetFormatEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                     *handle, flexio_i2s_format_t *format, uint32_t
                                     srcClock_Hz)
```

Configures the FlexIO I2S Tx audio format.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to format.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer
- format – Pointer to FlexIO I2S audio data format structure.
- srcClock_Hz – FlexIO I2S clock source frequency in Hz, it should be 0 while in slave mode.

```
status_t FLEXIO_I2S_TransferSendEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                     *handle, flexio_i2s_transfer_t *xfer)
```

Performs a non-blocking FlexIO I2S transfer using DMA.

Note: This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetTransferStatus to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- kStatus_Success – Start a FlexIO I2S eDMA send successfully.
- kStatus_InvalidArgument – The input arguments is invalid.
- kStatus_TxBusy – FlexIO I2S is busy sending data.

```
status_t FLEXIO_I2S_TransferReceiveEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                         *handle, flexio_i2s_transfer_t *xfer)
```

Performs a non-blocking FlexIO I2S receive using eDMA.

Note: This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetReceiveRemainingBytes to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.

- xfer – Pointer to DMA transfer structure.

Return values

- kStatus__Success – Start a FlexIO I2S eDMA receive successfully.
- kStatus__InvalidArgument – The input arguments is invalid.
- kStatus__RxBusy – FlexIO I2S is busy receiving data.

```
void FLEXIO_I2S_TransferAbortSendEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t *handle)
```

Aborts a FlexIO I2S transfer using eDMA.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.

```
void FLEXIO_I2S_TransferAbortReceiveEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t *handle)
```

Aborts a FlexIO I2S receive using eDMA.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.

```
status_t FLEXIO_I2S_TransferGetSendCountEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t *handle, size_t *count)
```

Gets the remaining bytes to be sent.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- count – Bytes sent.

Return values

- kStatus__Success – Succeed get the transfer count.
- kStatus__NoTransferInProgress – There is not a non-blocking transaction currently in progress.

```
status_t FLEXIO_I2S_TransferGetReceiveCountEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t *handle, size_t *count)
```

Get the remaining bytes to be received.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- count – Bytes received.

Return values

- kStatus__Success – Succeed get the transfer count.
- kStatus__NoTransferInProgress – There is not a non-blocking transaction currently in progress.

FSL_FLEXIO_I2S_EDMA_DRIVER_VERSION

FlexIO I2S EDMA driver version 2.1.9.

typedef struct *flexio_i2s_edma_handle* flexio_i2s_edma_handle_t

typedef void (*flexio_i2s_edma_callback_t)(FLEXIO_I2S_Type *base, *flexio_i2s_edma_handle_t* *handle, *status_t* status, void *userData)

FlexIO I2S eDMA transfer callback function for finish and error.

struct *flexio_i2s_edma_handle*

#include <fsl_flexio_i2s_edma.h> FlexIO I2S DMA transfer handle, users should not touch the content of the handle.

Public Members

edma_handle_t *dmaHandle

DMA handler for FlexIO I2S send

uint8_t bytesPerFrame

Bytes in a frame

uint8_t nbytes

eDMA minor byte transfer count initially configured.

uint32_t state

Internal state for FlexIO I2S eDMA transfer

flexio_i2s_edma_callback_t callback

Callback for users while transfer finish or error occurred

void *userData

User callback parameter

edma_tcd_t tcd[(4U) + 1U]

TCD pool for eDMA transfer.

flexio_i2s_transfer_t queue[(4U)]

Transfer queue storing queued transfer.

size_t transferSize[(4U)]

Data bytes need to transfer

volatile uint8_t queueUser

Index for user to queue transfer.

volatile uint8_t queueDriver

Index for driver to get the transfer data and size

2.18 FlexIO eDMA SPI Driver

status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA(*FLEXIO_SPI_Type* *base,
 flexio_spi_master_edma_handle_t
 *handle,
 flexio_spi_master_edma_transfer_callback_t
 callback, void *userData,
 edma_handle_t *txHandle,
 edma_handle_t *rxHandle)

Initializes the FlexIO SPI master eDMA handle.

This function initializes the FlexIO SPI master eDMA handle which can be used for other FlexIO SPI master transactional APIs. For a specified FlexIO SPI instance, call this API once to get the initialized handle.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

```
status_t FLEXIO_SPI_MasterTransferEDMA(FLEXIO_SPI_Type *base,  
                                         flexio_spi_master_edma_handle_t *handle,  
                                         flexio_spi_transfer_t *xfer)
```

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call FLEXIO_SPI_MasterGetTransferCountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – FlexIO SPI is not idle, is running another transfer.

```
void FLEXIO_SPI_MasterTransferAbortEDMA(FLEXIO_SPI_Type *base,  
                                         flexio_spi_master_edma_handle_t *handle)
```

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – FlexIO SPI eDMA handle pointer.

```
status_t FLEXIO_SPI_MasterTransferGetCountEDMA(FLEXIO_SPI_Type *base,  
                                              flexio_spi_master_edma_handle_t *handle,  
                                              size_t *count)
```

Gets the number of bytes transferred so far using FlexIO SPI master eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – FlexIO SPI eDMA handle pointer.
- count – Number of bytes transferred so far by the non-blocking transaction.

```
static inline void FLEXIO_SPI_SlaveTransferCreateHandleEDMA(FLEXIO_SPI_Type *base,  
                                                         flexio_spi_slave_edma_handle_t  
                                                         *handle,  
                                                         flexio_spi_slave_edma_transfer_callback_t  
                                                         callback, void *userData,  
                                                         edma_handle_t *txHandle,  
                                                         edma_handle_t *rxHandle)
```

Initializes the FlexIO SPI slave eDMA handle.

This function initializes the FlexIO SPI slave eDMA handle.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.

```
status_t FLEXIO_SPI_SlaveTransferEDMA(FLEXIO_SPI_Type *base,  
                                     flexio_spi_slave_edma_handle_t *handle,  
                                     flexio_spi_transfer_t *xfer)
```

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call FLEXIO_SPI_SlaveGetTransferCountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – FlexIO SPI is not idle, is running another transfer.

```
static inline void FLEXIO_SPI_SlaveTransferAbortEDMA(FLEXIO_SPI_Type *base,
                                                    flexio_spi_slave_edma_handle_t
                                                    *handle)
```

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – Pointer to *flexio_spi_slave_edma_handle_t* structure to store the transfer state.

```
static inline status_t FLEXIO_SPI_SlaveTransferGetCountEDMA(FLEXIO_SPI_Type *base,
                                                            flexio_spi_slave_edma_handle_t
                                                            *handle, size_t *count)
```

Gets the number of bytes transferred so far using FlexIO SPI slave eDMA.

Parameters

- base – Pointer to *FLEXIO_SPI_Type* structure.
- handle – FlexIO SPI eDMA handle pointer.
- count – Number of bytes transferred so far by the non-blocking transaction.

```
FSL_FLEXIO_SPI_EDMA_DRIVER_VERSION
```

FlexIO SPI EDMA driver version.

```
typedef struct flexio_spi_master_edma_handle flexio_spi_master_edma_handle_t
typedef for flexio_spi_master_edma_handle_t in advance.
```

```
typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t
Slave handle is the same with master handle.
```

```
typedef void (*flexio_spi_master_edma_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_master_edma_handle_t *handle, status_t status, void *userData)
```

FlexIO SPI master callback for finished transmit.

```
typedef void (*flexio_spi_slave_edma_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_slave_edma_handle_t *handle, status_t status, void *userData)
```

FlexIO SPI slave callback for finished transmit.

```
struct flexio_spi_master_edma_handle
```

#include <fsl_flexio_spi_edma.h> FlexIO SPI eDMA transfer handle, users should not touch the content of the handle.

Public Members

```
size_t transferSize
```

Total bytes to be transferred.

```
uint8_t nbytes
```

eDMA minor byte transfer count initially configured.

```
bool txInProgress
```

Send transfer in progress

```
bool rxInProgress
```

Receive transfer in progress

```
edma_handle_t *txHandle
```

DMA handler for SPI send

edma_handle_t *rxHandle
DMA handler for SPI receive

flexio_spi_master_edma_transfer_callback_t callback
Callback for SPI DMA transfer

void *userData
User Data for SPI DMA callback

2.19 FlexIO eDMA UART Driver

status_t FLEXIO_UART_TransferCreateHandleEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle,
 flexio_uart_edma_transfer_callback_t
 callback, void *userData, *edma_handle_t*
 *txEdmaHandle, *edma_handle_t*
 *rxEdmaHandle)

Initializes the UART handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_UART_Type.
- handle – Pointer to flexio_uart_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxEdmaHandle – User requested DMA handle for RX DMA transfer.
- txEdmaHandle – User requested DMA handle for TX DMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

status_t FLEXIO_UART_TransferSendEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle,
 flexio_uart_transfer_t *xfer)

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent out, the send callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – UART handle pointer.
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXIO_UART_TxBusy – Previous transfer on going.

status_t FLEXIO_UART_TransferReceiveEDMA(*FLEXIO_UART_Type* *base,
 flexio_uart_edma_handle_t *handle,
 flexio_uart_transfer_t *xfer)

Receives data using eDMA.

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus__Success – if succeed, others failed.
- kStatus__UART__RxBusy – Previous transfer on going.

```
void FLEXIO_UART_TransferAbortSendEDMA(FLEXIO_UART_Type *base,  
                                         flexio_uart_edma_handle_t *handle)
```

Aborts the sent data which using eDMA.

This function aborts sent data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

```
void FLEXIO_UART_TransferAbortReceiveEDMA(FLEXIO_UART_Type *base,  
                                           flexio_uart_edma_handle_t *handle)
```

Aborts the receive data which using eDMA.

This function aborts the receive data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

```
status_t FLEXIO_UART_TransferGetSendCountEDMA(FLEXIO_UART_Type *base,  
                                              flexio_uart_edma_handle_t *handle,  
                                              size_t *count)
```

Gets the number of bytes sent out.

This function gets the number of bytes sent out.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- count – Number of bytes sent so far by the non-blocking transaction.

Return values

- kStatus__NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus__Success – Successfully return the count.

```
status_t FLEXIO_UART_TransferGetReceiveCountEDMA(FLEXIO_UART_Type *base,  
                                                  flexio_uart_edma_handle_t *handle,  
                                                  size_t *count)
```

Gets the number of bytes received.

This function gets the number of bytes received.

Parameters

- `base` – Pointer to `FLEXIO_UART_Type`
- `handle` – Pointer to `flexio_uart_edma_handle_t` structure
- `count` – Number of bytes received so far by the non-blocking transaction.

Return values

- `kStatus_NoTransferInProgress` – transfer has finished or no transfer in progress.
- `kStatus_Success` – Successfully return the count.

`FSL_FLEXIO_UART_EDMA_DRIVER_VERSION`

FlexIO UART EDMA driver version.

`typedef struct flexio_uart_edma_handle flexio_uart_edma_handle_t`

`typedef void (*flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base,
flexio_uart_edma_handle_t *handle, status_t status, void *userData)`

UART transfer callback function.

`struct _flexio_uart_edma_handle`

`#include <fsl_flexio_uart_edma.h>` UART eDMA handle.

Public Members

`flexio_uart_edma_transfer_callback_t` callback

Callback function.

`void *userData`

UART callback function parameter.

`size_t txDataSizeAll`

Total bytes to be sent.

`size_t rxDataSizeAll`

Total bytes to be received.

`edma_handle_t *txEdmaHandle`

The eDMA TX channel used.

`edma_handle_t *rxEdmaHandle`

The eDMA RX channel used.

`uint8_t nbytes`

eDMA minor byte transfer count initially configured.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.20 FlexIO I2C Master Driver

status_t FLEXIO_I2C_CheckForBusyBus(*FLEXIO_I2C_Type* *base)

Make sure the bus isn't already pulled down.

Check the FLEXIO pin status to see whether either of SDA and SCL pin is pulled down.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure..

Return values

- kStatus_Success –
- kStatus_FLEXIO_I2C_Busy –

status_t FLEXIO_I2C_MasterInit(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.

Example

```
FLEXIO_I2C_Type base = {
    .flexioBase = FLEXIO,
    .SDAPinIndex = 0,
    .SCLPinIndex = 1,
    .shifterIndex = {0,1},
    .timerIndex = {0,1}
};
flexio_i2c_master_config_t config = {
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false,
    .baudRate_Bps = 100000
};
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- masterConfig – Pointer to flexio_i2c_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Initialization successful
- kStatus_InvalidArgument – The source clock exceed upper range limitation

void FLEXIO_I2C_MasterDeinit(*FLEXIO_I2C_Type* *base)

De-initializes the FlexIO I2C master peripheral. Calling this API Resets the FlexIO I2C master shifter and timer config, module can't work unless the FLEXIO_I2C_MasterInit is called.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterGetDefaultConfig(*flexio_i2c_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO module. The configuration can be used directly for calling the FLEXIO_I2C_MasterInit().

Example:

```
flexio_i2c_master_config_t config;  
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – Pointer to flexio_i2c_master_config_t structure.

static inline void FLEXIO_I2C_MasterEnable(*FLEXIO_I2C_Type* *base, bool enable)
Enables/disables the FlexIO module operation.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – Pass true to enable module, false does not have any effect.

uint32_t FLEXIO_I2C_MasterGetStatusFlags(*FLEXIO_I2C_Type* *base)
Gets the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure

Returns

Status flag, use status flag to AND _flexio_i2c_master_status_flags can get the related status.

void FLEXIO_I2C_MasterClearStatusFlags(*FLEXIO_I2C_Type* *base, uint32_t mask)
Clears the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
 - kFLEXIO_I2C_RxFullFlag
 - kFLEXIO_I2C_ReceiveNakFlag

void FLEXIO_I2C_MasterEnableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)
Enables the FlexIO i2c master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source. Currently only one interrupt request source:
 - kFLEXIO_I2C_TransferCompleteInterruptEnable

void FLEXIO_I2C_MasterDisableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)
Disables the FlexIO I2C master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source.

void FLEXIO_I2C_MasterSetBaudRate(*FLEXIO_I2C_Type* *base, uint32_t baudRate_Bps,
uint32_t srcClock_Hz)

Sets the FlexIO I2C master transfer baudrate.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- baudRate_Bps – the baud rate value in HZ

- srcClock_Hz – source clock in HZ

void FLEXIO_I2C_MasterStart(*FLEXIO_I2C_Type* *base, uint8_t address, *flexio_i2c_direction_t* direction)

Sends START + 7-bit address to the bus.

Note: This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO_I2C_RxFullFlag status is asserted before calling this API.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- address – 7-bit address.
- direction – transfer direction. This parameter is one of the values in flexio_i2c_direction_t:
 - kFLEXIO_I2C_Write: Transmit
 - kFLEXIO_I2C_Read: Receive

void FLEXIO_I2C_MasterStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterRepeatedStart(*FLEXIO_I2C_Type* *base)

Sends the repeated start signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterAbortStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal when transfer is still on-going.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterEnableAck(*FLEXIO_I2C_Type* *base, bool enable)

Configures the sent ACK/NAK for the following byte.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – True to configure send ACK, false configure to send NAK.

status_t FLEXIO_I2C_MasterSetTransferCount(*FLEXIO_I2C_Type* *base, uint16_t count)

Sets the number of bytes to be transferred from a start signal to a stop signal.

Note: Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

- `count` – Number of bytes need to be transferred from a start signal to a re-start/stop signal

Return values

- `kStatus_Success` – Successfully configured the count.
- `kStatus_InvalidArgument` – Input argument is invalid.

`static inline void FLEXIO_I2C_MasterWriteByte(FLEXIO_I2C_Type *base, uint32_t data)`

Writes one byte of data to the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the `TxEEmptyFlag` is asserted before calling this API.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `data` – a byte of data.

`static inline uint8_t FLEXIO_I2C_MasterReadByte(FLEXIO_I2C_Type *base)`

Reads one byte of data from the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.

Returns

data byte read.

`status_t FLEXIO_I2C_MasterWriteBlocking(FLEXIO_I2C_Type *base, const uint8_t *txBuff, uint8_t txSize)`

Sends a buffer of data in bytes.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `txBuff` – The data bytes to send.
- `txSize` – The number of data bytes to send.

Return values

- `kStatus_Success` – Successfully write data.
- `kStatus_FLEXIO_I2C_Nak` – Receive NAK during writing data.
- `kStatus_FLEXIO_I2C_Timeout` – Timeout polling status flags.

`status_t FLEXIO_I2C_MasterReadBlocking(FLEXIO_I2C_Type *base, uint8_t *rxBuff, uint8_t rxSize)`

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- rxBuff – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus__Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

status_t FLEXIO_I2C_MasterTransferBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.
- xfer – pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

status_t FLEXIO_I2C_MasterTransferCreateHandle(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_callback_t
callback, void *userData)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure to store the transfer state.
- callback – Pointer to user callback function.
- userData – User param passed to the callback function.

Return values

- kStatus__Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/isr table out of range.

status_t FLEXIO_I2C_MasterTransferNonBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_t *xfer)

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: The API returns immediately after the transfer initiates. Call FLEXIO_I2C_MasterTransferGetCount to poll the transfer status to check whether the

transfer is finished. If the return status is not `kStatus_FLEXIO_I2C_Busy`, the transfer is finished.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure
- `handle` – Pointer to `flexio_i2c_master_handle_t` structure which stores the transfer state
- `xfer` – pointer to `flexio_i2c_master_transfer_t` structure

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_FLEXIO_I2C_Busy` – FlexIO I2C is not idle, is running another transfer.

`status_t FLEXIO_I2C_MasterTransferGetCount(FLEXIO_I2C_Type *base,
flexio_i2c_master_handle_t *handle, size_t
*count)`

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `handle` – Pointer to `flexio_i2c_master_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.
- `kStatus_Success` – Successfully return the count.

`void FLEXIO_I2C_MasterTransferAbort(FLEXIO_I2C_Type *base, flexio_i2c_master_handle_t
*handle)`

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure
- `handle` – Pointer to `flexio_i2c_master_handle_t` structure which stores the transfer state

`void FLEXIO_I2C_MasterTransferHandleIRQ(void *i2cType, void *i2cHandle)`

Master interrupt handler.

Parameters

- `i2cType` – Pointer to `FLEXIO_I2C_Type` structure
- `i2cHandle` – Pointer to `flexio_i2c_master_transfer_t` structure

FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION

FlexIO I2C transfer status.

Values:

enumerator kStatus_FLEXIO_I2C_Busy

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Idle

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Nak

NAK received during transfer.

enumerator kStatus_FLEXIO_I2C_Timeout

Timeout polling status flags.

enum _flexio_i2c_master_interrupt

Define FlexIO I2C master interrupt mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyInterruptEnable

Tx buffer empty interrupt enable.

enumerator kFLEXIO_I2C_RxFullInterruptEnable

Rx buffer full interrupt enable.

enum _flexio_i2c_master_status_flags

Define FlexIO I2C master status mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyFlag

Tx shifter empty flag.

enumerator kFLEXIO_I2C_RxFullFlag

Rx shifter full/Transfer complete flag.

enumerator kFLEXIO_I2C_ReceiveNakFlag

Receive NAK flag.

enum _flexio_i2c_direction

Direction of master transfer.

Values:

enumerator kFLEXIO_I2C_Write

Master send to slave.

enumerator kFLEXIO_I2C_Read

Master receive from slave.

typedef enum _flexio_i2c_direction flexio_i2c_direction_t

Direction of master transfer.

typedef struct _flexio_i2c_type FLEXIO_I2C_Type

Define FlexIO I2C master access structure typedef.

typedef struct _flexio_i2c_master_config flexio_i2c_master_config_t

Define FlexIO I2C master user configuration structure.

```
typedef struct _flexio_i2c_master_transfer flexio_i2c_master_transfer_t
```

Define FlexIO I2C master transfer structure.

```
typedef struct _flexio_i2c_master_handle flexio_i2c_master_handle_t
```

FlexIO I2C master handle typedef.

```
typedef void (*flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type *base,  
flexio_i2c_master_handle_t *handle, status_t status, void *userData)
```

FlexIO I2C master transfer callback typedef.

```
I2C_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _flexio_i2c_type
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t SDAPinIndex

Pin select for I2C SDA.

uint8_t SCLPinIndex

Pin select for I2C SCL.

uint8_t shifterIndex[2]

Shifter index used in FlexIO I2C.

uint8_t timerIndex[3]

Timer index used in FlexIO I2C.

uint32_t baudrate

Master transfer baudrate, used to calculate delay time.

```
struct _flexio_i2c_master_config
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master user configuration structure.

Public Members

bool enableMaster

Enables the FlexIO I2C peripheral at initialization time.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

```
struct _flexio_i2c_master_transfer
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master transfer structure.

Public Members

uint32_t flags

Transfer flag which controls the transfer, reserved for FlexIO I2C.

uint8_t slaveAddress

7-bit slave address.

flexio_i2c_direction_t direction

Transfer direction, read or write.

uint32_t subaddress

Sub address. Transferred MSB first.

uint8_t subaddressSize

Size of sub address.

uint8_t volatile *data

Transfer buffer.

volatile size_t dataSize

Transfer size.

struct *_flexio_i2c_master_handle*

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master handle structure.

Public Members

flexio_i2c_master_transfer_t transfer

FlexIO I2C master transfer copy.

size_t transferSize

Total bytes to be transferred.

uint8_t state

Transfer state maintained during transfer.

flexio_i2c_master_transfer_callback_t completionCallback

Callback function called at transfer event. Callback function called at transfer event.

void *userData

Callback parameter passed to callback function.

bool needRestart

Whether master needs to send re-start signal.

2.21 FlexIO I2S Driver

void FLEXIO_I2S_Init(*FLEXIO_I2S_Type* *base, const *flexio_i2s_config_t* *config)

Initializes the FlexIO I2S.

This API configures FlexIO pins and shifter to I2S and configures the FlexIO I2S with a configuration structure. The configuration structure can be filled by the user, or be set with default values by FLEXIO_I2S_GetDefaultConfig().

Note: This API should be called at the beginning of the application to use the FlexIO I2S driver. Otherwise, any access to the FlexIO I2S module can cause hard fault because the clock is not enabled.

Parameters

- base – FlexIO I2S base pointer
- config – FlexIO I2S configure structure.

void FLEXIO_I2S_GetDefaultConfig(*flexio_i2s_config_t* *config)

Sets the FlexIO I2S configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in FLEXIO_I2S_Init(). Users may use the initialized structure unchanged in FLEXIO_I2S_Init() or modify some fields of the structure before calling FLEXIO_I2S_Init().

Parameters

- config – pointer to master configuration structure

void FLEXIO_I2S_Deinit(*FLEXIO_I2S_Type* *base)

De-initializes the FlexIO I2S.

Calling this API resets the FlexIO I2S shifter and timer config. After calling this API, call the FLEXIO_I2S_Init to use the FlexIO I2S module.

Parameters

- base – FlexIO I2S base pointer

static inline void FLEXIO_I2S_Enable(*FLEXIO_I2S_Type* *base, bool enable)

Enables/disables the FlexIO I2S module operation.

Parameters

- base – Pointer to FLEXIO_I2S_Type
- enable – True to enable, false dose not have any effect.

uint32_t FLEXIO_I2S_GetStatusFlags(*FLEXIO_I2S_Type* *base)

Gets the FlexIO I2S status flags.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure

Returns

Status flag, which are ORed by the enumerators in the `_flexio_i2s_status_flags`.

void FLEXIO_I2S_EnableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Enables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

void FLEXIO_I2S_DisableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Disables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

static inline void FLEXIO_I2S_TxEnableDMA(*FLEXIO_I2S_Type* *base, bool enable)
Enables/disables the FlexIO I2S Tx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

static inline void FLEXIO_I2S_RxEnableDMA(*FLEXIO_I2S_Type* *base, bool enable)
Enables/disables the FlexIO I2S Rx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t FLEXIO_I2S_TxGetDataRegisterAddress(*FLEXIO_I2S_Type* *base)
Gets the FlexIO I2S send data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure

Returns

FlexIO i2s send data register address.

static inline uint32_t FLEXIO_I2S_RxGetDataRegisterAddress(*FLEXIO_I2S_Type* *base)
Gets the FlexIO I2S receive data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure

Returns

FlexIO i2s receive data register address.

void FLEXIO_I2S_MasterSetFormat(*FLEXIO_I2S_Type* *base, *flexio_i2s_format_t* *format,
uint32_t srcClock_Hz)

Configures the FlexIO I2S audio format in master mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- format – Pointer to FlexIO I2S audio data format structure.
- srcClock_Hz – I2S master clock source frequency in Hz.

void FLEXIO_I2S_SlaveSetFormat(*FLEXIO_I2S_Type* *base, *flexio_i2s_format_t* *format)
Configures the FlexIO I2S audio format in slave mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- format – Pointer to FlexIO I2S audio data format structure.

```
status_t FLEXIO_I2S_WriteBlocking(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint8_t *txData,  
                                size_t size)
```

Sends data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- txData – Pointer to the data to be written.
- size – Bytes to be written.

Return values

- kStatus_Success – Successfully write data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

```
static inline void FLEXIO_I2S_WriteData(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint32_t  
                                       data)
```

Writes data into a data register.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- data – Data to be written.

```
status_t FLEXIO_I2S_ReadBlocking(FLEXIO_I2S_Type *base, uint8_t bitWidth, uint8_t *rxData,  
                                size_t size)
```

Receives a piece of data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- rxData – Pointer to the data to be read.
- size – Bytes to be read.

Return values

- kStatus_Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

```
static inline uint32_t FLEXIO_I2S_ReadData(FLEXIO_I2S_Type *base)
```

Reads a data from the data register.

Parameters

- base – FlexIO I2S base pointer

Returns

Data read from data register.

```
void FLEXIO_I2S_TransferTxCreateHandle(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                       flexio_i2s_callback_t callback, void *userData)
```

Initializes the FlexIO I2S handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure
- handle – Pointer to *flexio_i2s_handle_t* structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

```
void FLEXIO_I2S_TransferSetFormat(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                  flexio_i2s_format_t *format, uint32_t srcClock_Hz)
```

Configures the FlexIO I2S audio format.

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.
- handle – FlexIO I2S handle pointer.
- format – Pointer to audio data format structure.
- srcClock_Hz – FlexIO I2S bit clock source frequency in Hz. This parameter should be 0 while in slave mode.

```
void FLEXIO_I2S_TransferRxCreateHandle(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
                                       flexio_i2s_callback_t callback, void *userData)
```

Initializes the FlexIO I2S receive handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.
- handle – Pointer to *flexio_i2s_handle_t* structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

```
status_t FLEXIO_I2S_TransferSendNonBlocking(FLEXIO_I2S_Type *base, flexio_i2s_handle_t  
                                             *handle, flexio_i2s_transfer_t *xfer)
```

Performs an interrupt non-blocking send transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call *FLEXIO_I2S_GetRemainingBytes* to poll the transfer status and check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- base – Pointer to *FLEXIO_I2S_Type* structure.

- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `xfer` – Pointer to `flexio_i2s_transfer_t` structure

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_FLEXIO_I2S_TxBusy` – Previous transmission still not finished, data not all written to TX register yet.
- `kStatus_InvalidArgument` – The input parameter is invalid.

`status_t FLEXIO_I2S_TransferReceiveNonBlocking(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_transfer_t *xfer)`

Performs an interrupt non-blocking receive transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call `FLEXIO_I2S_GetRemainingBytes` to poll the transfer status to check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `xfer` – Pointer to `flexio_i2s_transfer_t` structure

Return values

- `kStatus_Success` – Successfully start the data receive.
- `kStatus_FLEXIO_I2S_RxBusy` – Previous receive still not finished.
- `kStatus_InvalidArgument` – The input parameter is invalid.

`void FLEXIO_I2S_TransferAbortSend(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle)`

Aborts the current send.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state

`void FLEXIO_I2S_TransferAbortReceive(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle)`

Aborts the current receive.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.

- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state

`status_t FLEXIO_I2S_TransferGetSendCount(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`

Gets the remaining bytes to be sent.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `count` – Bytes sent.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`status_t FLEXIO_I2S_TransferGetReceiveCount(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`

Gets the remaining bytes to be received.

Parameters

- `base` – Pointer to `FLEXIO_I2S_Type` structure.
- `handle` – Pointer to `flexio_i2s_handle_t` structure which stores the transfer state
- `count` – Bytes recieved.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

Returns

`count` Bytes received.

`void FLEXIO_I2S_TransferTxHandleIRQ(void *i2sBase, void *i2sHandle)`

Tx interrupt handler.

Parameters

- `i2sBase` – Pointer to `FLEXIO_I2S_Type` structure.
- `i2sHandle` – Pointer to `flexio_i2s_handle_t` structure

`void FLEXIO_I2S_TransferRxHandleIRQ(void *i2sBase, void *i2sHandle)`

Rx interrupt handler.

Parameters

- `i2sBase` – Pointer to `FLEXIO_I2S_Type` structure.
- `i2sHandle` – Pointer to `flexio_i2s_handle_t` structure.

`FSL_FLEXIO_I2S_DRIVER_VERSION`

FlexIO I2S driver version 2.2.2.

FlexIO I2S transfer status.

Values:

enumerator kStatus_FLEXIO_I2S_Idle

FlexIO I2S is in idle state

enumerator kStatus_FLEXIO_I2S_TxBusy

FlexIO I2S Tx is busy

enumerator kStatus_FLEXIO_I2S_RxBusy

FlexIO I2S Rx is busy

enumerator kStatus_FLEXIO_I2S_Error

FlexIO I2S error occurred

enumerator kStatus_FLEXIO_I2S_QueueFull

FlexIO I2S transfer queue is full.

enumerator kStatus_FLEXIO_I2S_Timeout

FlexIO I2S timeout polling status flags.

enum _flexio_i2s_master_slave

Master or slave mode.

Values:

enumerator kFLEXIO_I2S_Master

Master mode

enumerator kFLEXIO_I2S_Slave

Slave mode

_flexio_i2s_interrupt_enable Define FlexIO FlexIO I2S interrupt mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyInterruptEnable

Transmit buffer empty interrupt enable.

enumerator kFLEXIO_I2S_RxDataRegFullInterruptEnable

Receive buffer full interrupt enable.

_flexio_i2s_status_flags Define FlexIO FlexIO I2S status mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyFlag

Transmit buffer empty flag.

enumerator kFLEXIO_I2S_RxDataRegFullFlag

Receive buffer full flag.

enum _flexio_i2s_sample_rate

Audio sample rate.

Values:

enumerator kFLEXIO_I2S_SampleRate8KHz

Sample rate 8000Hz

```
enumerator kFLEXIO_I2S_SampleRate11025Hz
    Sample rate 11025Hz
enumerator kFLEXIO_I2S_SampleRate12KHz
    Sample rate 12000Hz
enumerator kFLEXIO_I2S_SampleRate16KHz
    Sample rate 16000Hz
enumerator kFLEXIO_I2S_SampleRate22050Hz
    Sample rate 22050Hz
enumerator kFLEXIO_I2S_SampleRate24KHz
    Sample rate 24000Hz
enumerator kFLEXIO_I2S_SampleRate32KHz
    Sample rate 32000Hz
enumerator kFLEXIO_I2S_SampleRate44100Hz
    Sample rate 44100Hz
enumerator kFLEXIO_I2S_SampleRate48KHz
    Sample rate 48000Hz
enumerator kFLEXIO_I2S_SampleRate96KHz
    Sample rate 96000Hz
enum _flexio_i2s_word_width
    Audio word width.
    Values:
enumerator kFLEXIO_I2S_WordWidth8bits
    Audio data width 8 bits
enumerator kFLEXIO_I2S_WordWidth16bits
    Audio data width 16 bits
enumerator kFLEXIO_I2S_WordWidth24bits
    Audio data width 24 bits
enumerator kFLEXIO_I2S_WordWidth32bits
    Audio data width 32 bits
typedef struct _flexio_i2s_type FLEXIO_I2S_Type
    Define FlexIO I2S access structure typedef.
typedef enum _flexio_i2s_master_slave flexio_i2s_master_slave_t
    Master or slave mode.
typedef struct _flexio_i2s_config flexio_i2s_config_t
    FlexIO I2S configure structure.
typedef struct _flexio_i2s_format flexio_i2s_format_t
    FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.
typedef enum _flexio_i2s_sample_rate flexio_i2s_sample_rate_t
    Audio sample rate.
typedef enum _flexio_i2s_word_width flexio_i2s_word_width_t
    Audio word width.
```

```
typedef struct _flexio_i2s_transfer flexio_i2s_transfer_t
```

Define FlexIO I2S transfer structure.

```
typedef struct _flexio_i2s_handle flexio_i2s_handle_t
```

```
typedef void (*flexio_i2s_callback_t)(FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle,  
status_t status, void *userData)
```

FlexIO I2S xfer callback prototype.

```
I2S_RETRY_TIMES
```

Retry times for waiting flag.

```
FLEXIO_I2S_XFER_QUEUE_SIZE
```

FlexIO I2S transfer queue size, user can refine it according to use case.

```
struct _flexio_i2s_type
```

#include <fsl_flexio_i2s.h> Define FlexIO I2S access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer

uint8_t txPinIndex

Tx data pin index in FlexIO pins

uint8_t rxPinIndex

Rx data pin index

uint8_t bclkPinIndex

Bit clock pin index

uint8_t fsPinIndex

Frame sync pin index

uint8_t txShifterIndex

Tx data shifter index

uint8_t rxShifterIndex

Rx data shifter index

uint8_t bclkTimerIndex

Bit clock timer index

uint8_t fsTimerIndex

Frame sync timer index

```
struct _flexio_i2s_config
```

#include <fsl_flexio_i2s.h> FlexIO I2S configure structure.

Public Members

bool enableI2S

Enable FlexIO I2S

flexio_i2s_master_slave_t masterSlave

Master or slave

flexio_pin_polarity_t txPinPolarity

Tx data pin polarity, active high or low

flexio_pin_polarity_t rxPinPolarity
Rx data pin polarity

flexio_pin_polarity_t bclkPinPolarity
Bit clock pin polarity

flexio_pin_polarity_t fsPinPolarity
Frame sync pin polarity

flexio_shifter_timer_polarity_t txTimerPolarity
Tx data valid on bclk rising or falling edge

flexio_shifter_timer_polarity_t rxTimerPolarity
Rx data valid on bclk rising or falling edge

struct *_flexio_i2s_format*

#include <fsl_flexio_i2s.h> FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.

Public Members

uint8_t bitWidth
Bit width of audio data, always 8/16/24/32 bits

uint32_t sampleRate_Hz
Sample rate of the audio data

struct *_flexio_i2s_transfer*

#include <fsl_flexio_i2s.h> Define FlexIO I2S transfer structure.

Public Members

uint8_t *data
Data buffer start pointer

size_t dataSize
Bytes to be transferred.

struct *_flexio_i2s_handle*

#include <fsl_flexio_i2s.h> Define FlexIO I2S handle structure.

Public Members

uint32_t state
Internal state

flexio_i2s_callback_t callback
Callback function called at transfer event

void *userData
Callback parameter passed to callback function

uint8_t bitWidth
Bit width for transfer, 8/16/24/32bits

flexio_i2s_transfer_t queue[(4U)]
Transfer queue storing queued transfer

size_t transferSize[(4U)]
 Data bytes need to transfer

volatile uint8_t queueUser
 Index for user to queue transfer

volatile uint8_t queueDriver
 Index for driver to get the transfer data and size

2.22 FlexIO SPI Driver

void FLEXIO_SPI_MasterInit(*FLEXIO_SPI_Type* *base, *flexio_spi_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI master hardware, and configures the FlexIO SPI with FlexIO SPI master configuration. The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_MasterGetDefaultConfig().

Example

```
FLEXIO_SPI_Type spiDev = {  
    .flexioBase = FLEXIO,  
    .SDOPinIndex = 0,  
    .SDIPinIndex = 1,  
    .SCKPinIndex = 2,  
    .CSnPinIndex = 3,  
    .shifterIndex = {0,1},  
    .timerIndex = {0,1}  
};  
flexio_spi_master_config_t config = {  
    .enableMaster = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false,  
    .baudRate_Bps = 500000,  
    .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,  
    .direction = kFLEXIO_SPI_MsbFirst,  
    .dataMode = kFLEXIO_SPI_8BitMode  
};  
FLEXIO_SPI_MasterInit(&spiDev, &config, srcClock_Hz);
```

Note: 1.FlexIO SPI master only support CPOL = 0, which means clock inactive low. 2.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI master communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by 2*2=4. If FlexIO SPI master communicates with FlexIO SPI slave, the maximum baud rate is FlexIO clock frequency divided by (1.5+2.5)*2=8.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- masterConfig – Pointer to the flexio_spi_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

void FLEXIO_SPI_MasterDeinit(*FLEXIO_SPI_Type* *base)

Resets the FlexIO SPI timer and shifter config.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_MasterGetDefaultConfig(*flexio_spi_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO SPI master. The configuration can be used directly by calling the FLEXIO_SPI_MasterConfigure(). Example:

```
flexio_spi_master_config_t masterConfig;
FLEXIO_SPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – Pointer to the flexio_spi_master_config_t structure.

void FLEXIO_SPI_SlaveInit(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_config_t* *slaveConfig)

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI slave hardware configuration, and configures the FlexIO SPI with FlexIO SPI slave configuration. The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_SlaveGetDefaultConfig().

Note: 1. Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. 2. FlexIO SPI slave only support CPOL = 0, which means clock inactive low. 3. For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI slave communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by $3 \times 2 = 6$. If FlexIO SPI slave communicates with FlexIO SPI master, the maximum baud rate is FlexIO clock frequency divided by $(1.5 + 2.5) \times 2 = 8$. Example

```
FLEXIO_SPI_Type spiDev = {
    .flexioBase = FLEXIO,
    .SDOPinIndex = 0,
    .SDIPinIndex = 1,
    .SCKPinIndex = 2,
    .CSnPinIndex = 3,
    .shifterIndex = {0,1},
    .timerIndex = {0}
};
flexio_spi_slave_config_t config = {
    .enableSlave = true,
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false,
    .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
    .direction = kFLEXIO_SPI_MsbFirst,
    .dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_SlaveInit(&spiDev, &config);
```

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

void FLEXIO_SPI_SlaveDeinit(*FLEXIO_SPI_Type* *base)

Gates the FlexIO clock.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_SlaveGetDefaultConfig(*flexio_spi_slave_config_t* *slaveConfig)

Gets the default configuration to configure the FlexIO SPI slave. The configuration can be used directly for calling the FLEXIO_SPI_SlaveConfigure(). Example:

```
flexio_spi_slave_config_t slaveConfig;  
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

uint32_t FLEXIO_SPI_GetStatusFlags(*FLEXIO_SPI_Type* *base)

Gets FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- kFLEXIO_SPI_TxEmptyFlag
- kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_ClearStatusFlags(*FLEXIO_SPI_Type* *base, uint32_t mask)

Clears FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – status flag The parameter can be any combination of the following values:
 - kFLEXIO_SPI_TxEmptyFlag
 - kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_EnableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Enables the FlexIO SPI interrupt.

This function enables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source. The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_DisableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Disables the FlexIO SPI interrupt.

This function disables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_EnableDMA(*FLEXIO_SPI_Type* *base, uint32_t mask, bool enable)

Enables/disables the FlexIO SPI transmit DMA. This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO_SPI_TxEmptyFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – SPI DMA source.
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t FLEXIO_SPI_GetTxDataRegisterAddress(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction)

Gets the FlexIO SPI transmit data register address for MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI transmit data register address.

static inline uint32_t FLEXIO_SPI_GetRxDataRegisterAddress(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction)

Gets the FlexIO SPI receive data register address for the MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI receive data register address.

static inline void FLEXIO_SPI_Enable(*FLEXIO_SPI_Type* *base, bool enable)

Enables/disables the FlexIO SPI module operation.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.
- enable – True to enable, false does not have any effect.

void FLEXIO_SPI_MasterSetBaudRate(*FLEXIO_SPI_Type* *base, uint32_t baudRate_Bps, uint32_t srcClockHz)

Sets baud rate for the FlexIO SPI transfer, which is only used for the master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- baudRate_Bps – Baud Rate needed in Hz.
- srcClockHz – SPI source clock frequency in Hz.

```
static inline void FLEXIO_SPI_WriteData(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                         direction, uint32_t data)
```

Writes one byte of data, which is sent using the MSB method.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- data – 8/16/32 bit data.

```
static inline uint32_t FLEXIO_SPI_ReadData(FLEXIO_SPI_Type *base,  
                                           flexio_spi_shift_direction_t direction)
```

Reads 8 bit/16 bit data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

8 bit/16 bit data received.

```
status_t FLEXIO_SPI_WriteBlocking(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                   direction, const uint8_t *buffer, size_t size)
```

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The data bytes to send.
- size – The number of data bytes to send.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

```
status_t FLEXIO_SPI_ReadBlocking(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t
                                direction, uint8_t *buffer, size_t size)
```

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The buffer to store the received bytes.
- size – The number of data bytes to be received.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

```
status_t FLEXIO_SPI_MasterTransferBlocking(FLEXIO_SPI_Type *base, flexio_spi_transfer_t
                                           *xfer)
```

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – pointer to *FLEXIO_SPI_Type* structure
- xfer – FlexIO SPI transfer structure, see *flexio_spi_transfer_t*.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

```
void FLEXIO_SPI_FlushShifters(FLEXIO_SPI_Type *base)
```

Flush tx/rx shifters.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.

```
status_t FLEXIO_SPI_MasterTransferCreateHandle(FLEXIO_SPI_Type *base,
                                              flexio_spi_master_handle_t *handle,
                                              flexio_spi_master_transfer_callback_t
                                              callback, void *userData)
```

Initializes the FlexIO SPI Master handle, which is used in transactional functions.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- handle – Pointer to the *flexio_spi_master_handle_t* structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- `kStatus__Success` – Successfully create the handle.
- `kStatus__OutOfRange` – The FlexIO type/handle/ISR table out of range.

`status_t` FLEXIO_SPI_MasterTransferNonBlocking(*FLEXIO_SPI_Type* *base,
flexio_spi_master_handle_t *handle,
flexio_spi_transfer_t *xfer)

Master transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.
- `xfer` – FlexIO SPI transfer structure. See `flexio_spi_transfer_t`.

Return values

- `kStatus__Success` – Successfully start a transfer.
- `kStatus__InvalidArgument` – Input argument is invalid.
- `kStatus__FLEXIO_SPI_Busy` – SPI is not idle, is running another transfer.

`void` FLEXIO_SPI_MasterTransferAbort(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle)

Aborts the master data transfer, which used IRQ.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.

`status_t` FLEXIO_SPI_MasterTransferGetCount(*FLEXIO_SPI_Type* *base,
flexio_spi_master_handle_t *handle, *size_t* *count)

Gets the data transfer status which used IRQ.

Parameters

- `base` – Pointer to the `FLEXIO_SPI_Type` structure.
- `handle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus__InvalidArgument` – `count` is Invalid.
- `kStatus__Success` – Successfully return the count.

`void` FLEXIO_SPI_MasterTransferHandleIRQ(*void* *spiType, *void* *spiHandle)

FlexIO SPI master IRQ handler function.

Parameters

- `spiType` – Pointer to the `FLEXIO_SPI_Type` structure.
- `spiHandle` – Pointer to the `flexio_spi_master_handle_t` structure to store the transfer state.


```
status_t FLEXIO_SPI_SlaveTransferCreateHandle(FLEXIO_SPI_Type *base,
                                             flexio_spi_slave_handle_t *handle,
                                             flexio_spi_slave_transfer_callback_t callback,
                                             void *userData)
```

Initializes the FlexIO SPI Slave handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

```
status_t FLEXIO_SPI_SlaveTransferNonBlocking(FLEXIO_SPI_Type *base,
                                             flexio_spi_slave_handle_t *handle,
                                             flexio_spi_transfer_t *xfer)
```

Slave transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- base – Pointer to the FLEXIO_SPI_Type structure.
- xfer – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – SPI is not idle; it is running another transfer.

```
static inline void FLEXIO_SPI_SlaveTransferAbort(FLEXIO_SPI_Type *base,
                                                flexio_spi_slave_handle_t *handle)
```

Aborts the slave data transfer which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

```
static inline status_t FLEXIO_SPI_SlaveTransferGetCount(FLEXIO_SPI_Type *base,
                                                       flexio_spi_slave_handle_t *handle,
                                                       size_t *count)
```

Gets the data transfer status which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void FLEXIO_SPI_SlaveTransferHandleIRQ(void *spiType, void *spiHandle)`

FlexIO SPI slave IRQ handler function.

Parameters

- `spiType` – Pointer to the `FLEXIO_SPI_Type` structure.
- `spiHandle` – Pointer to the `flexio_spi_slave_handle_t` structure to store the transfer state.

`FSL_FLEXIO_SPI_DRIVER_VERSION`

FlexIO SPI driver version.

Error codes for the FlexIO SPI driver.

Values:

enumerator `kStatus_FLEXIO_SPI_Busy`
FlexIO SPI is busy.

enumerator `kStatus_FLEXIO_SPI_Idle`
SPI is idle

enumerator `kStatus_FLEXIO_SPI_Error`
FlexIO SPI error.

enumerator `kStatus_FLEXIO_SPI_Timeout`
FlexIO SPI timeout polling status flags.

enum `_flexio_spi_clock_phase`

FlexIO SPI clock phase configuration.

Values:

enumerator `kFLEXIO_SPI_ClockPhaseFirstEdge`
First edge on SPCK occurs at the middle of the first cycle of a data transfer.

enumerator `kFLEXIO_SPI_ClockPhaseSecondEdge`
First edge on SPCK occurs at the start of the first cycle of a data transfer.

enum `_flexio_spi_shift_direction`

FlexIO SPI data shifter direction options.

Values:

enumerator `kFLEXIO_SPI_MsbFirst`
Data transfers start with most significant bit.

enumerator `kFLEXIO_SPI_LsbFirst`
Data transfers start with least significant bit.

enum `_flexio_spi_data_bitcount_mode`

FlexIO SPI data length mode options.

Values:

enumerator `kFLEXIO_SPI_8BitMode`
8-bit data transmission mode.

enumerator kFLEXIO_SPI_16BitMode
16-bit data transmission mode.

enumerator kFLEXIO_SPI_32BitMode
32-bit data transmission mode.

enum _flexio_spi_interrupt_enable
Define FlexIO SPI interrupt mask.

Values:

enumerator kFLEXIO_SPI_TxEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_SPI_RxFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_spi_status_flags
Define FlexIO SPI status mask.

Values:

enumerator kFLEXIO_SPI_TxBufferEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_SPI_RxBufferFullFlag
Receive buffer full flag.

enum _flexio_spi_dma_enable
Define FlexIO SPI DMA mask.

Values:

enumerator kFLEXIO_SPI_TxDmaEnable
Tx DMA request source

enumerator kFLEXIO_SPI_RxDmaEnable
Rx DMA request source

enumerator kFLEXIO_SPI_DmaAllEnable
All DMA request source

enum _flexio_spi_transfer_flags
Define FlexIO SPI transfer flags.

Note: Use kFLEXIO_SPI_csContinuous and one of the other flags to OR together to form the transfer flag.

Values:

enumerator kFLEXIO_SPI_8bitMsb
FlexIO SPI 8-bit MSB first

enumerator kFLEXIO_SPI_8bitLsb
FlexIO SPI 8-bit LSB first

enumerator kFLEXIO_SPI_16bitMsb
FlexIO SPI 16-bit MSB first

enumerator kFLEXIO_SPI_16bitLsb
FlexIO SPI 16-bit LSB first

enumerator kFLEXIO__SPI__32bitMsb
FlexIO SPI 32-bit MSB first

enumerator kFLEXIO__SPI__32bitLsb
FlexIO SPI 32-bit LSB first

enumerator kFLEXIO__SPI__csContinuous
Enable the CS signal continuous mode

typedef enum *_flexio_spi_clock_phase* flexio_spi_clock_phase_t
FlexIO SPI clock phase configuration.

typedef enum *_flexio_spi_shift_direction* flexio_spi_shift_direction_t
FlexIO SPI data shifter direction options.

typedef enum *_flexio_spi_data_bitcount_mode* flexio_spi_data_bitcount_mode_t
FlexIO SPI data length mode options.

typedef struct *_flexio_spi_type* FLEXIO__SPI_Type
Define FlexIO SPI access structure typedef.

typedef struct *_flexio_spi_master_config* flexio_spi_master_config_t
Define FlexIO SPI master configuration structure.

typedef struct *_flexio_spi_slave_config* flexio_spi_slave_config_t
Define FlexIO SPI slave configuration structure.

typedef struct *_flexio_spi_transfer* flexio_spi_transfer_t
Define FlexIO SPI transfer structure.

typedef struct *_flexio_spi_master_handle* flexio_spi_master_handle_t
typedef for flexio_spi_master_handle_t in advance.

typedef *flexio_spi_master_handle_t* flexio_spi_slave_handle_t
Slave handle is the same with master handle.

typedef void (*flexio_spi_master_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_master_handle_t *handle, *status_t* status, void *userData)
FlexIO SPI master callback for finished transmit.

typedef void (*flexio_spi_slave_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_slave_handle_t *handle, *status_t* status, void *userData)
FlexIO SPI slave callback for finished transmit.

FLEXIO__SPI__DUMMYDATA
FlexIO SPI dummy transfer data, the data is sent while txData is NULL.

SPI_RETRY_TIMES
Retry times for waiting flag.

FLEXIO_SPI_XFER_DATA_FORMAT(flag)
Get the transfer data format of width and bit order.

struct *_flexio_spi_type*
#include <fsl_flexio_spi.h> Define FlexIO SPI access structure typedef.

Public Members

FLEXIO_Type *flexioBase
FlexIO base pointer.

uint8_t SDOPinIndex

Pin select for data output. To set SDO pin in Hi-Z state, user needs to mux the pin as GPIO input and disable all pull up/down in application.

uint8_t SDIPinIndex

Pin select for data input.

uint8_t SCKPinIndex

Pin select for clock.

uint8_t CSnPinIndex

Pin select for enable.

uint8_t shifterIndex[2]

Shifter index used in FlexIO SPI.

uint8_t timerIndex[2]

Timer index used in FlexIO SPI.

struct __flexio_spi_master_config

#include <fsl_flexio_spi.h> Define FlexIO SPI master configuration structure.

Public Members

bool enableMaster

Enable/disable FlexIO SPI master after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct __flexio_spi_slave_config

#include <fsl_flexio_spi.h> Define FlexIO SPI slave configuration structure.

Public Members

bool enableSlave

Enable/disable FlexIO SPI slave after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct *_flexio_spi_transfer*

#include <fsl_flexio_spi.h> Define FlexIO SPI transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

size_t dataSize

Transfer bytes.

uint8_t flags

FlexIO SPI control flag, MSB first or LSB first.

struct *_flexio_spi_master_handle*

#include <fsl_flexio_spi.h> Define FlexIO SPI handle structure.

Public Members

const uint8_t *txData

Transfer buffer.

uint8_t *rxData

Receive buffer.

size_t transferSize

Total bytes to be transferred.

volatile size_t txRemainingBytes

Send data remaining in bytes.

volatile size_t rxRemainingBytes

Receive data remaining in bytes.

volatile uint32_t state

FlexIO SPI internal state.

uint8_t bytePerFrame

SPI mode, 2bytes or 1byte in a frame

flexio_spi_shift_direction_t direction

Shift direction.

flexio_spi_master_transfer_callback_t callback

FlexIO SPI callback.

`void *userData`
 Callback parameter.

`bool isCsContinuous`
 Is current transfer using CS continuous mode.

`uint32_t timer1Cfg`
 TIMER1 TIMCFG regiser value backup.

2.23 FlexIO UART Driver

`status_t FLEXIO_UART_Init(FLEXIO_UART_Type *base, const flexio_uart_config_t *userConfig, uint32_t srcClock_Hz)`

Ungates the FlexIO clock, resets the FlexIO module, configures FlexIO UART hardware, and configures the FlexIO UART with FlexIO UART configuration. The configuration structure can be filled by the user or be set with default values by `FLEXIO_UART_GetDefaultConfig()`.

Example

```
FLEXIO_UART_Type base = {
    .flexioBase = FLEXIO,
    .TxPinIndex = 0,
    .RxPinIndex = 1,
    .shifterIndex = {0,1},
    .timerIndex = {0,1}
};
flexio_uart_config_t config = {
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false,
    .baudRate_Bps = 115200U,
    .bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `userConfig` – Pointer to the `flexio_uart_config_t` structure.
- `srcClock_Hz` – FlexIO source clock in Hz.

Return values

- `kStatus_Success` – Configuration success.
- `kStatus_FLEXIO_UART_BaudrateNotSupport` – Baudrate is not supported for current clock source frequency.

`void FLEXIO_UART_Deinit(FLEXIO_UART_Type *base)`
 Resets the FlexIO UART shifter and timer config.

Note: After calling this API, call the `FLEXIO_UART_Init` to use the FlexIO UART module.

Parameters

- `base` – Pointer to `FLEXIO_UART_Type` structure

void FLEXIO_UART_GetDefaultConfig(*flexio_uart_config_t* *userConfig)

Gets the default configuration to configure the FlexIO UART. The configuration can be used directly for calling the FLEXIO_UART_Init(). Example:

```
flexio_uart_config_t config;  
FLEXIO_UART_GetDefaultConfig(&userConfig);
```

Parameters

- userConfig – Pointer to the flexio_uart_config_t structure.

uint32_t FLEXIO_UART_GetStatusFlags(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART status flags.

void FLEXIO_UART_ClearStatusFlags(*FLEXIO_UART_Type* *base, uint32_t mask)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
 - kFLEXIO_UART_TxDataRegEmptyFlag
 - kFLEXIO_UART_RxEmptyFlag
 - kFLEXIO_UART_RxOverRunFlag

void FLEXIO_UART_EnableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Enables the FlexIO UART interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

void FLEXIO_UART_DisableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Disables the FlexIO UART interrupt.

This function disables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

static inline uint32_t FLEXIO_UART_GetTxDataRegisterAddress(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART transmit data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART transmit data register address.

```
static inline uint32_t FLEXIO_UART_GetRxDataRegisterAddress(FLEXIO_UART_Type *base)
```

Gets the FlexIO UART receive data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.

Returns

FlexIO UART receive data register address.

```
static inline void FLEXIO_UART_EnableTxDMA(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART transmit DMA. This function enables/disables the FlexIO UART Tx DMA, which means asserting the *kFLEXIO_UART_TxDataRegEmptyFlag* does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- enable – True to enable, false to disable.

```
static inline void FLEXIO_UART_EnableRxDMA(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART receive DMA. This function enables/disables the FlexIO UART Rx DMA, which means asserting *kFLEXIO_UART_RxDataRegFullFlag* does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- enable – True to enable, false to disable.

```
static inline void FLEXIO_UART_Enable(FLEXIO_UART_Type *base, bool enable)
```

Enables/disables the FlexIO UART module operation.

Parameters

- base – Pointer to the *FLEXIO_UART_Type*.
- enable – True to enable, false does not have any effect.

```
static inline void FLEXIO_UART_WriteByte(FLEXIO_UART_Type *base, const uint8_t *buffer)
```

Writes one byte of data.

Note: This is a non-blocking API, which returns directly after the data is put into the data register. Ensure that the *TxEmptyFlag* is asserted before calling this API.

Parameters

- base – Pointer to the *FLEXIO_UART_Type* structure.
- buffer – The data bytes to send.

```
static inline void FLEXIO_UART_ReadByte(FLEXIO_UART_Type *base, uint8_t *buffer)
```

Reads one byte of data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the *RxFullFlag* is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- buffer – The buffer to store the received bytes.

status_t FLEXIO_UART_WriteBlocking(*FLEXIO_UART_Type* *base, const uint8_t *txData, size_t txSize)

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- txData – The data bytes to send.
- txSize – The number of data bytes to send.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t FLEXIO_UART_ReadBlocking(*FLEXIO_UART_Type* *base, uint8_t *rxData, size_t rxSize)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- rxData – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t FLEXIO_UART_TransferCreateHandle(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *flexio_uart_transfer_callback_t* callback, void *userData)

Initializes the UART handle.

This function initializes the FlexIO UART handle, which can be used for other FlexIO UART transactional APIs. Call this API once to get the initialized handle.

The UART driver supports the “background” receiving, which means that users can set up a RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn’t call the FLEXIO_UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as ringBuffer.

Parameters

- base – to FLEXIO_UART_Type structure.

- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `callback` – The callback function.
- `userData` – The parameter of the callback function.

Return values

- `kStatus_Success` – Successfully create the handle.
- `kStatus_OutOfRange` – The FlexIO type/handle/ISR table out of range.

```
void FLEXIO_UART_TransferStartRingBuffer(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                         *handle, uint8_t *ringBuffer, size_t
                                         ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the `UART_ReceiveNonBlocking()` API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `ringBuffer` – Start address of ring buffer for background receiving. Pass `NULL` to disable the ring buffer.
- `ringBufferSize` – Size of the ring buffer.

```
void FLEXIO_UART_TransferStopRingBuffer(FLEXIO_UART_Type *base, flexio_uart_handle_t
                                         *handle)
```

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.

```
status_t FLEXIO_UART_TransferSendNonBlocking(FLEXIO_UART_Type *base,
                                              flexio_uart_handle_t *handle,
                                              flexio_uart_transfer_t *xfer)
```

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in ISR, the FlexIO UART driver calls the callback function and passes the `kStatus_FLEXIO_UART_TxIdle` as status parameter.

Note: The `kStatus_FLEXIO_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `xfer` – FlexIO UART transfer structure. See `flexio_uart_transfer_t`.

Return values

- `kStatus_Success` – Successfully starts the data transmission.
- `kStatus_UART_TxBusy` – Previous transmission still not finished, data not written to the TX register.

```
void FLEXIO_UART_TransferAbortSend(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle)
```

Aborts the interrupt-driven data transmit.

This function aborts the interrupt-driven data sending. Get the `remainBytes` to find out how many bytes are still not sent out.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.

```
status_t FLEXIO_UART_TransferGetSendCount(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, size_t *count)
```

Gets the number of bytes sent.

This function gets the number of bytes sent driven by interrupt.

Parameters

- `base` – Pointer to the `FLEXIO_UART_Type` structure.
- `handle` – Pointer to the `flexio_uart_handle_t` structure to store the transfer state.
- `count` – Number of bytes sent so far by the non-blocking transaction.

Return values

- `kStatus_NoTransferInProgress` – transfer has finished or no transfer in progress.
- `kStatus_Success` – Successfully return the count.

```
status_t FLEXIO_UART_TransferReceiveNonBlocking(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, flexio_uart_transfer_t *xfer, size_t *receivedBytes)
```

Receives a buffer of data using the interrupt method.

This function receives data using the interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in ring buffer is not enough to read, the receive request is saved by the UART driver. When new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_UART_RxIdle`. For example, if the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer, the 5 bytes are copied to `xfer->data`. This function returns with the parameter `receivedBytes` set to 5. For the last 5 bytes, newly arrived data is saved from the

xfer->data[5]. When 5 bytes are received, the UART driver notifies upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to xfer->data. When all data is received, the upper layer is notified.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- xfer – UART transfer structure. See flexio_uart_transfer_t.
- receivedBytes – Bytes received from the ring buffer directly.

Return values

- kStatus__Success – Successfully queue the transfer into the transmit queue.
- kStatus_FLEXIO_UART_RxBusy – Previous receive request is not finished.

```
void FLEXIO_UART_TransferAbortReceive(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle)
```

Aborts the receive data which was using IRQ.

This function aborts the receive data which was using IRQ.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

```
status_t FLEXIO_UART_TransferGetReceiveCount(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, size_t *count)
```

Gets the number of bytes received.

This function gets the number of bytes received driven by interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- count – Number of bytes received so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus__Success – Successfully return the count.

```
void FLEXIO_UART_TransferHandleIRQ(void *uartType, void *uartHandle)
```

FlexIO UART IRQ handler function.

This function processes the FlexIO UART transmit and receives the IRQ request.

Parameters

- uartType – Pointer to the FLEXIO_UART_Type structure.
- uartHandle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

void FLEXIO_UART_FlushShifters(*FLEXIO_UART_Type* *base)

Flush tx/rx shifters.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

FSL_FLEXIO_UART_DRIVER_VERSION

FlexIO UART driver version.

Error codes for the UART driver.

Values:

enumerator kStatus_FLEXIO_UART_TxBusy

Transmitter is busy.

enumerator kStatus_FLEXIO_UART_RxBusy

Receiver is busy.

enumerator kStatus_FLEXIO_UART_TxIdle

UART transmitter is idle.

enumerator kStatus_FLEXIO_UART_RxIdle

UART receiver is idle.

enumerator kStatus_FLEXIO_UART_ERROR

ERROR happens on UART.

enumerator kStatus_FLEXIO_UART_RxRingBufferOverrun

UART RX software ring buffer overrun.

enumerator kStatus_FLEXIO_UART_RxHardwareOverrun

UART RX receiver overrun.

enumerator kStatus_FLEXIO_UART_Timeout

UART times out.

enumerator kStatus_FLEXIO_UART_BaudrateNotSupport

Baudrate is not supported in current clock source

enum _flexio_uart_bit_count_per_char

FlexIO UART bit count per char.

Values:

enumerator kFLEXIO_UART_7BitsPerChar

7-bit data characters

enumerator kFLEXIO_UART_8BitsPerChar

8-bit data characters

enumerator kFLEXIO_UART_9BitsPerChar

9-bit data characters

enum _flexio_uart_interrupt_enable

Define FlexIO UART interrupt mask.

Values:

enumerator kFLEXIO_UART_TxDataRegEmptyInterruptEnable

Transmit buffer empty interrupt enable.

```

    enumerator kFLEXIO_UART_RxDataRegFullInterruptEnable
        Receive buffer full interrupt enable.

enum _flexio_uart_status_flags
    Define FlexIO UART status mask.

    Values:

    enumerator kFLEXIO_UART_TxDataRegEmptyFlag
        Transmit buffer empty flag.

    enumerator kFLEXIO_UART_RxDataRegFullFlag
        Receive buffer full flag.

    enumerator kFLEXIO_UART_RxOverRunFlag
        Receive buffer over run flag.

typedef enum _flexio_uart_bit_count_per_char flexio_uart_bit_count_per_char_t
    FlexIO UART bit count per char.

typedef struct _flexio_uart_type FLEXIO_UART_Type
    Define FlexIO UART access structure typedef.

typedef struct _flexio_uart_config flexio_uart_config_t
    Define FlexIO UART user configuration structure.

typedef struct _flexio_uart_transfer flexio_uart_transfer_t
    Define FlexIO UART transfer structure.

typedef struct _flexio_uart_handle flexio_uart_handle_t

typedef void (*flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t
*handle, status_t status, void *userData)
    FlexIO UART transfer callback function.

UART_RETRY_TIMES
    Retry times for waiting flag.

struct _flexio_uart_type
    #include <fsl_flexio_uart.h> Define FlexIO UART access structure typedef.

```

Public Members

```

FLEXIO_Type *flexioBase
    FlexIO base pointer.

uint8_t TxPinIndex
    Pin select for UART_Tx.

uint8_t RxPinIndex
    Pin select for UART_Rx.

uint8_t shifterIndex[2]
    Shifter index used in FlexIO UART.

uint8_t timerIndex[2]
    Timer index used in FlexIO UART.

struct _flexio_uart_config
    #include <fsl_flexio_uart.h> Define FlexIO UART user configuration structure.

```

Public Members

`bool enableUart`
Enable/disable FlexIO UART TX & RX.

`bool enableInDoze`
Enable/disable FlexIO operation in doze mode

`bool enableInDebug`
Enable/disable FlexIO operation in debug mode

`bool enableFastAccess`
Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

`uint32_t baudRate_Bps`
Baud rate in Bps.

`flexio_uart_bit_count_per_char_t bitCountPerChar`
number of bits, 7/8/9 -bit

`struct _flexio_uart_transfer`
#include <fsl_flexio_uart.h> Define FlexIO UART transfer structure.

Public Members

`size_t dataSize`
Transfer size

`struct _flexio_uart_handle`
#include <fsl_flexio_uart.h> Define FLEXIO UART handle structure.

Public Members

`const uint8_t *volatile txData`
Address of remaining data to send.

`volatile size_t txDataSize`
Size of the remaining data to send.

`uint8_t *volatile rxData`
Address of remaining data to receive.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t txDataSizeAll`
Total bytes to be sent.

`size_t rxDataSizeAll`
Total bytes to be received.

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
Index for the user to get data from the ring buffer.

flexio_uart_transfer_callback_t callback
Callback function.

void *userData
UART callback function parameter.

volatile uint8_t txState
TX transfer state.

volatile uint8_t rxState
RX transfer state

union __unnamed43__

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.

2.24 FLEXRAM: on-chip RAM manager

void FLEXRAM_Init(FLEXRAM_Type *base)
FLEXRAM module initialization function.

Parameters

- base – FLEXRAM base address.

void FLEXRAM_Deinit(FLEXRAM_Type *base)
De-initializes the FLEXRAM.

static inline uint32_t FLEXRAM_GetInterruptStatus(FLEXRAM_Type *base)
FLEXRAM module gets interrupt status.

Parameters

- base – FLEXRAM base address.

static inline void FLEXRAM_ClearInterruptStatus(FLEXRAM_Type *base, uint32_t status)
FLEXRAM module clears interrupt status.

Parameters

- base – FLEXRAM base address.
- status – Status to be cleared.

static inline void FLEXRAM_EnableInterruptStatus(FLEXRAM_Type *base, uint32_t status)
FLEXRAM module enables interrupt status.

Parameters

- base – FLEXRAM base address.

- status – Status to be enabled.

static inline void FLEXRAM_DisableInterruptStatus(FLEXRAM_Type *base, uint32_t status)
FLEXRAM module disable interrupt status.

Parameters

- base – FLEXRAM base address.
- status – Status to be disabled.

static inline void FLEXRAM_EnableInterruptSignal(FLEXRAM_Type *base, uint32_t status)
FLEXRAM module enables interrupt.

Parameters

- base – FLEXRAM base address.
- status – Status interrupt to be enabled.

static inline void FLEXRAM_DisableInterruptSignal(FLEXRAM_Type *base, uint32_t status)
FLEXRAM module disables interrupt.

Parameters

- base – FLEXRAM base address.
- status – Status interrupt to be disabled.

FSL_FLEXRAM_DRIVER_VERSION
Driver version.

Flexram write/read selection.

Values:

enumerator kFLEXRAM_Read
read

enumerator kFLEXRAM_Write
write

Interrupt status flag mask.

Values:

enumerator kFLEXRAM_OCRAMAccessError
OCRAM accesses unallocated address

enumerator kFLEXRAM_DTCMAccessError
DTCM accesses unallocated address

enumerator kFLEXRAM_ITCMAccessError
ITCM accesses unallocated address

enumerator kFLEXRAM_OCRAMMagicAddrMatch
OCRAM magic address match

enumerator kFLEXRAM_DTCMMagicAddrMatch
DTCM magic address match

enumerator kFLEXRAM_ITCMMagicAddrMatch
ITCM magic address match

enumerator kFLEXRAM_OCRAMECCMultiError

enumerator kFLEXRAM_OCAMECCSingleError

enumerator kFLEXRAM_ITCMECCMultiError

enumerator kFLEXRAM_ITCMECCSingleError

enumerator kFLEXRAM_D0TCMECCMultiError

enumerator kFLEXRAM_D0TCMECCSingleError

enumerator kFLEXRAM_D1TCMECCMultiError

enumerator kFLEXRAM_D1TCMECCSingleError

enumerator kFLEXRAM_InterruptStatusAll

enum _flexram_tcm_access_mode

FLEXRAM TCM access mode. Fast access mode expected to be finished in 1-cycle; Wait access mode expected to be finished in 2-cycle. Wait access mode is a feature of the flexram and it should be used when the CPU clock is too fast to finish TCM access in 1-cycle. Normally, fast mode is the default mode, the efficiency of the TCM access will better.

Values:

enumerator kFLEXRAM_TCMAccessFastMode

fast access mode

enumerator kFLEXRAM_TCMAccessWaitMode

wait access mode

FLEXRAM TCM support size.

Values:

enumerator kFLEXRAM_TCMSize32KB

TCM total size be 32KB

enumerator kFLEXRAM_TCMSize64KB

TCM total size be 64KB

enumerator kFLEXRAM_TCMSize128KB

TCM total size be 128KB

enumerator kFLEXRAM_TCMSize256KB

TCM total size be 256KB

enumerator kFLEXRAM_TCMSize512KB

TCM total size be 512KB

enum _flexram_memory_type

FLEXRAM memory type, such as OCRAM/ITCM/D0TCM/D1TCM.

Values:

enumerator kFLEXRAM_OCRAM

Memory type OCRAM

enumerator kFLEXRAM_ITCM

Memory type ITCM

enumerator kFLEXRAM_D0TCM

Memory type D0TCM

enumerator kFLEXRAM_D1TCM

Memory type D1TCM

typedef enum *flexram_tcm_access_mode* flexram_tcm_access_mode_t

FLEXRAM TCM access mode. Fast access mode expected to be finished in 1-cycle; Wait access mode expected to be finished in 2-cycle. Wait access mode is a feature of the flexram and it should be used when the CPU clock is too fast to finish TCM access in 1-cycle. Normally, fast mode is the default mode, the efficiency of the TCM access will better.

typedef enum *flexram_memory_type* flexram_memory_type_t

FLEXRAM memory type, such as OCRAM/ITCM/D0TCM/D1TCM.

typedef struct *flexram_ecc_error_type* flexram_ecc_error_type_t

FLEXRAM error type, such as single bit error position, multi-bit error position.

typedef struct *flexram_ocram_ecc_single_error_info* flexram_ocram_ecc_single_error_info_t

FLEXRAM ocram ecc single error information, including single error information, error address, error data.

typedef struct *flexram_ocram_ecc_multi_error_info* flexram_ocram_ecc_multi_error_info_t

FLEXRAM ocram ecc multiple error information, including multiple error information, error address, error data.

typedef struct *flexram_itcm_ecc_single_error_info* flexram_itcm_ecc_single_error_info_t

FLEXRAM itcm ecc single error information, including single error information, error address, error data.

typedef struct *flexram_itcm_ecc_multi_error_info* flexram_itcm_ecc_multi_error_info_t

FLEXRAM itcm ecc multiple error information, including multiple error information, error address, error data.

typedef struct *flexram_dtcmm_ecc_single_error_info* flexram_dtcmm_ecc_single_error_info_t

FLEXRAM dtcm ecc single error information, including single error information, error address, error data.

typedef struct *flexram_dtcmm_ecc_multi_error_info* flexram_dtcmm_ecc_multi_error_info_t

FLEXRAM dtcm ecc multiple error information, including multiple error information, error address, error data.

static inline void FLEXRAM_SetTCMReadAccessMode(FLEXRAM_Type *base,
flexram_tcm_access_mode_t mode)

FLEXRAM module sets TCM read access mode.

Parameters

- base – FLEXRAM base address.
- mode – Access mode.

static inline void FLEXRAM_SetTCMWriteAccessMode(FLEXRAM_Type *base,
flexram_tcm_access_mode_t mode)

FLEXRAM module set TCM write access mode.

Parameters

- base – FLEXRAM base address.
- mode – Access mode.

static inline void FLEXRAM_EnableForceRamClockOn(FLEXRAM_Type *base, bool enable)

FLEXRAM module force ram clock on.

Parameters

- base – FLEXRAM base address.

- enable – Enable or disable clock force on.

```
static inline void FLEXRAM_SetOCRAMMagicAddr(FLEXRAM_Type *base, uint16_t magicAddr,  
                                             uint32_t rwSel)
```

FLEXRAM OCRAM magic addr configuration. When read/write access hit magic address, it will generate interrupt.

Parameters

- base – FLEXRAM base address.
- magicAddr – Magic address, the actual address bits [18:3] is corresponding to the register field [16:1].
- rwSel – Read/write selection. 0 for read access while 1 for write access.

```
static inline void FLEXRAM_SetDTCMMagicAddr(FLEXRAM_Type *base, uint16_t magicAddr,  
                                             uint32_t rwSel)
```

FLEXRAM DTCM magic addr configuration. When read/write access hits magic address, it will generate interrupt.

Parameters

- base – FLEXRAM base address.
- magicAddr – Magic address, the actual address bits [18:3] is corresponding to the register field [16:1].
- rwSel – Read/write selection. 0 for read access while 1 write access.

```
static inline void FLEXRAM_SetITCMMagicAddr(FLEXRAM_Type *base, uint16_t magicAddr,  
                                             uint32_t rwSel)
```

FLEXRAM ITCM magic addr configuration. When read/write access hits magic address, it will generate interrupt.

Parameters

- base – FLEXRAM base address.
- magicAddr – Magic address, the actual address bits [18:3] is corresponding to the register field [16:1].
- rwSel – Read/write selection. 0 for read access while 1 for write access.

```
void FLEXRAM_EnableECC(FLEXRAM_Type *base, bool OcramECCEnable, bool  
                      TcmECCEnable)
```

FLEXRAM get ocram ecc single error information.

Parameters

- base – FLEXRAM base address.
- OcramECCEnable – ocram ecc enablement.
- TcmECCEnable – tcm(itcm/d0tcm/d1tcm) ecc enablement.

```
void FLEXRAM_ErrorInjection(FLEXRAM_Type *base, flexram_memory_type_t memory,  
                           flexram_ecc_error_type_t *error)
```

FLEXRAM ECC error injection.

Parameters

- base – FLEXRAM base address.
- memory – memory type, such as OCRAM/ITCM/DTCM.
- error – ECC error type.

```
void FLEXRAM_GetOcramSingleErrInfo(FLEXRAM_Type *base,  
                                   flexram_ocram_ecc_single_error_info_t *info)
```

FLEXRAM get ocram ecc single error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.

```
void FLEXRAM_GetOcramMultiErrInfo(FLEXRAM_Type *base,  
                                   flexram_ocram_ecc_multi_error_info_t *info)
```

FLEXRAM get ocram ecc multiple error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.

```
void FLEXRAM_GetItcmSingleErrInfo(FLEXRAM_Type *base,  
                                   flexram_itcm_ecc_single_error_info_t *info)
```

FLEXRAM get itcm ecc single error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.

```
void FLEXRAM_GetItcmMultiErrInfo(FLEXRAM_Type *base,  
                                   flexram_itcm_ecc_multi_error_info_t *info)
```

FLEXRAM get itcm ecc multiple error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.

```
void FLEXRAM_GetDtcSingleErrInfo(FLEXRAM_Type *base,  
                                   flexram_dtc_ecc_single_error_info_t *info, uint8_t  
                                   bank)
```

FLEXRAM get d0tcm ecc single error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.
- bank – DTCM bank, 0 is D0TCM, 1 is D1TCM.

```
void FLEXRAM_GetDtcMultiErrInfo(FLEXRAM_Type *base,  
                                   flexram_dtc_ecc_multi_error_info_t *info, uint8_t  
                                   bank)
```

FLEXRAM get d0tcm ecc multiple error information.

Parameters

- base – FLEXRAM base address.
- info – ecc error information.
- bank – DTCM bank, 0 is D0TCM, 1 is D1TCM.

```
FLEXRAM_ECC_ERROR_DETAILED_INFO
```

Get ECC error detailed information.

```
struct _flexram_ecc_error_type
    #include <fsl_flexram.h> FLEXRAM error type, such as single bit error position, multi-bit
    error position.
```

Public Members

```
uint8_t SingleBitPos
    Bit position of the bit to inject ECC Error.
uint8_t SecondBitPos
    Bit position of the second bit to inject multi-bit ECC Error
bool Fource1BitDataInversion
    Force One 1-Bit Data Inversion (single-bit ECC error) on memory write access
bool FourceOneNCDataInversion
    Force One Non-correctable Data Inversion(multi-bit ECC error) on memory write ac-
    cess
bool FourceConti1BitDataInversion
    Force Continuous 1-Bit Data Inversions (single-bit ECC error) on memory write access
bool FourceContiNCDataInversion
    Force Continuous Non-correctable Data Inversions (multi-bit ECC error) on memory
    write access
```

```
struct _flexram_ocram_ecc_single_error_info
    #include <fsl_flexram.h> FLEXRAM ocram ecc single error information, including single er-
    ror information, error address, error data.
```

Public Members

```
uint32_t OcramSingleErrorInfo
    Ocram single error information, user should parse it by themself.
uint32_t OcramSingleErrorAddr
    Ocram single error address
uint32_t OcramSingleErrorDataLSB
    Ocram single error data LSB
uint32_t OcramSingleErrorDataMSB
    Ocram single error data MSB
```

```
struct _flexram_ocram_ecc_multi_error_info
    #include <fsl_flexram.h> FLEXRAM ocram ecc multiple error information, including multi-
    ple error information, error address, error data.
```

Public Members

```
uint32_t OcramMultiErrorInfo
    Ocram single error information, user should parse it by themself.
uint32_t OcramMultiErrorAddr
    Ocram multiple error address
uint32_t OcramMultiErrorDataLSB
    Ocram multiple error data LSB
```

uint32_t OcramMultiErrorDataMSB
Ocram multiple error data MSB

struct _flexram_itcm_ecc_single_error_info
#include <fsl_flexram.h> FLEXRAM itcm ecc single error information, including single error information, error address, error data.

Public Members

uint32_t ItcmSingleErrorInfo
itcm single error information, user should parse it by themselves.

uint32_t ItcmSingleErrorAddr
itcm single error address

uint32_t ItcmSingleErrorDataLSB
itcm single error data LSB

uint32_t ItcmSingleErrorDataMSB
itcm single error data MSB

struct _flexram_itcm_ecc_multi_error_info
#include <fsl_flexram.h> FLEXRAM itcm ecc multiple error information, including multiple error information, error address, error data.

Public Members

uint32_t ItcmMultiErrorInfo
itcm multiple error information, user should parse it by themselves.

uint32_t ItcmMultiErrorAddr
itcm multiple error address

uint32_t ItcmMultiErrorDataLSB
itcm multiple error data LSB

uint32_t ItcmMultiErrorDataMSB
itcm multiple error data MSB

struct _flexram_dtcn_ecc_single_error_info
#include <fsl_flexram.h> FLEXRAM dtcm ecc single error information, including single error information, error address, error data.

Public Members

uint32_t DtcnSingleErrorInfo
dtcm single error information, user should parse it by themselves.

uint32_t DtcnSingleErrorAddr
dtcm single error address

uint32_t DtcnSingleErrorData
dtcm single error data

struct _flexram_dtcn_ecc_multi_error_info
#include <fsl_flexram.h> FLEXRAM dtcm ecc multiple error information, including multiple error information, error address, error data.

Public Members

uint32_t DtcMultiErrorInfo

dtcm multiple error information, user should parse it by themself.

uint32_t DtcMultiErrorAddr

dtcm multiple error address

uint32_t DtcMultiErrorData

dtcm multiple error data

2.25 FLEXSPI: Flexible Serial Peripheral Interface Driver

uint32_t FLEXSPI_GetInstance(FLEXSPI_Type *base)

Get the instance number for FLEXSPI.

Parameters

- base – FLEXSPI base pointer.

status_t FLEXSPI_CheckAndClearError(FLEXSPI_Type *base, uint32_t status)

Check and clear IP command execution errors.

Parameters

- base – FLEXSPI base pointer.
- status – interrupt status.

void FLEXSPI_Init(FLEXSPI_Type *base, const flexspi_config_t *config)

Initializes the FLEXSPI module and internal state.

This function enables the clock for FLEXSPI and also configures the FLEXSPI with the input configure parameters. Users should call this function before any FLEXSPI operations.

Parameters

- base – FLEXSPI peripheral base address.
- config – FLEXSPI configure structure.

void FLEXSPI_GetDefaultConfig(flexspi_config_t *config)

Gets default settings for FLEXSPI.

Parameters

- config – FLEXSPI configuration structure.

void FLEXSPI_Deinit(FLEXSPI_Type *base)

Deinitializes the FLEXSPI module.

Clears the FLEXSPI state and FLEXSPI module registers.

Parameters

- base – FLEXSPI peripheral base address.

void FLEXSPI_UpdateDllValue(FLEXSPI_Type *base, flexspi_device_config_t *config, flexspi_port_t port)

Update FLEXSPI DLL value depending on currently flexspi root clock.

Parameters

- base – FLEXSPI peripheral base address.
- config – Flash configuration parameters.

- port – FLEXSPI Operation port.

```
void FLEXSPI_SetFlashConfig(FLEXSPI_Type *base, flexspi_device_config_t *config,  
                           flexspi_port_t port)
```

Configures the connected device parameter.

This function configures the connected device relevant parameters, such as the size, command, and so on. The flash configuration value cannot have a default value. The user needs to configure it according to the connected device.

Parameters

- base – FLEXSPI peripheral base address.
- config – Flash configuration parameters.
- port – FLEXSPI Operation port.

```
void FLEXSPI_SoftwareReset(FLEXSPI_Type *base)
```

Software reset for the FLEXSPI logic.

This function sets the software reset flags for both AHB and buffer domain and resets both AHB buffer and also IP FIFOs.

Parameters

- base – FLEXSPI peripheral base address.

```
static inline void FLEXSPI_Enable(FLEXSPI_Type *base, bool enable)
```

Enables or disables the FLEXSPI module.

Parameters

- base – FLEXSPI peripheral base address.
- enable – True means enable FLEXSPI, false means disable.

```
void FLEXSPI_UpdateAhbBuffersSettings(FLEXSPI_Type *base, flexspi_ahbBuffers_ctrl_t  
                                     *ptrAhbBufferCtrl)
```

Update all AHB buffers' settings, including buffer size, master ID.

Parameters

- base – FLEXSPI peripheral base address.
- ptrAhbBufferCtrl – Pointer to structure flexspi_ahbBuffers_ctrl_t which store all AHB buffers' settings.

```
static inline void FLEXSPI_EnableInterrupts(FLEXSPI_Type *base, uint32_t mask)
```

Enables the FLEXSPI interrupts.

Parameters

- base – FLEXSPI peripheral base address.
- mask – FLEXSPI interrupt source.

```
static inline void FLEXSPI_DisableInterrupts(FLEXSPI_Type *base, uint32_t mask)
```

Disable the FLEXSPI interrupts.

Parameters

- base – FLEXSPI peripheral base address.
- mask – FLEXSPI interrupt source.

```
static inline void FLEXSPI_EnableTxDMA(FLEXSPI_Type *base, bool enable)
```

Enables or disables FLEXSPI IP Tx FIFO DMA requests.

Parameters

- base – FLEXSPI peripheral base address.
- enable – Enable flag for transmit DMA request. Pass true for enable, false for disable.

static inline void FLEXSPI_EnableRxDMA(FLEXSPI_Type *base, bool enable)

Enables or disables FLEXSPI IP Rx FIFO DMA requests.

Parameters

- base – FLEXSPI peripheral base address.
- enable – Enable flag for receive DMA request. Pass true for enable, false for disable.

static inline uint32_t FLEXSPI_GetTxFifoAddress(FLEXSPI_Type *base)

Gets FLEXSPI IP tx fifo address for DMA transfer.

Parameters

- base – FLEXSPI peripheral base address.

Return values

The – tx fifo address.

static inline uint32_t FLEXSPI_GetRxFifoAddress(FLEXSPI_Type *base)

Gets FLEXSPI IP rx fifo address for DMA transfer.

Parameters

- base – FLEXSPI peripheral base address.

Return values

The – rx fifo address.

static inline void FLEXSPI_ResetFifos(FLEXSPI_Type *base, bool txFifo, bool rxFifo)

Clears the FLEXSPI IP FIFO logic.

Parameters

- base – FLEXSPI peripheral base address.
- txFifo – Pass true to reset TX FIFO.
- rxFifo – Pass true to reset RX FIFO.

static inline void FLEXSPI_GetFifoCounts(FLEXSPI_Type *base, size_t *txCount, size_t *rxCount)

Gets the valid data entries in the FLEXSPI FIFOs.

Parameters

- base – FLEXSPI peripheral base address.
- txCount – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

static inline uint32_t FLEXSPI_GetInterruptStatusFlags(FLEXSPI_Type *base)

Get the FLEXSPI interrupt status flags.

Parameters

- base – FLEXSPI peripheral base address.

Return values

interrupt – status flag, use status flag to AND flexspi_flags_t could get the related status.

```
static inline void FLEXSPI_ClearInterruptStatusFlags(FLEXSPI_Type *base, uint32_t mask)
```

Get the FLEXSPI interrupt status flags.

Parameters

- base – FLEXSPI peripheral base address.
- mask – FLEXSPI interrupt source.

```
static inline flexspi_arb_command_source_t FLEXSPI_GetArbitratorCommandSource(FLEXSPI_Type *base)
```

Gets the trigger source of current command sequence granted by arbitrator.

Parameters

- base – FLEXSPI peripheral base address.

Return values

trigger – source of current command sequence.

```
static inline flexspi_ip_error_code_t FLEXSPI_GetIPCommandErrorCode(FLEXSPI_Type *base, uint8_t *index)
```

Gets the error code when IP command error detected.

Parameters

- base – FLEXSPI peripheral base address.
- index – Pointer to a uint8_t type variable to receive the sequence index when error detected.

Return values

error – code when IP command error detected.

```
static inline flexspi_ahb_error_code_t FLEXSPI_GetAHBCommandErrorCode(FLEXSPI_Type *base, uint8_t *index)
```

Gets the error code when AHB command error detected.

Parameters

- base – FLEXSPI peripheral base address.
- index – Pointer to a uint8_t type variable to receive the sequence index when error detected.

Return values

error – code when AHB command error detected.

```
static inline bool FLEXSPI_GetBusIdleStatus(FLEXSPI_Type *base)
```

Returns whether the bus is idle.

Parameters

- base – FLEXSPI peripheral base address.

Return values

- true – Bus is idle.
- false – Bus is busy.

```
void FLEXSPI_UpdateRxSampleClock(FLEXSPI_Type *base, flexspi_read_sample_clock_t clockSource)
```

Update read sample clock source.

Parameters

- base – FLEXSPI peripheral base address.

- clockSource – clockSource of type flexspi_read_sample_clock_t

void FLEXSPI_UpdateLUT(FLEXSPI_Type *base, uint32_t index, const uint32_t *cmd, uint32_t count)

Updates the LUT table.

Parameters

- base – FLEXSPI peripheral base address.
- index – From which index start to update. It could be any index of the LUT table, which also allows user to update command content inside a command. Each command consists of up to 8 instructions and occupy 4*32-bit memory.
- cmd – Command sequence array.
- count – Number of sequences.

static inline void FLEXSPI_WriteData(FLEXSPI_Type *base, uint32_t data, uint8_t fifoIndex)

Writes data into FIFO.

Parameters

- base – FLEXSPI peripheral base address
- data – The data bytes to send
- fifoIndex – Destination fifo index.

static inline uint32_t FLEXSPI_ReadData(FLEXSPI_Type *base, uint8_t fifoIndex)

Receives data from data FIFO.

Parameters

- base – FLEXSPI peripheral base address
- fifoIndex – Source fifo index.

Returns

The data in the FIFO.

status_t FLEXSPI_WriteBlocking(FLEXSPI_Type *base, uint8_t *buffer, size_t size)

Sends a buffer of data bytes using blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – FLEXSPI peripheral base address
- buffer – The data bytes to send
- size – The number of data bytes to send

Return values

- kStatus_Success – write success without error
- kStatus_FLEXSPI_SequenceExecutionTimeout – sequence execution timeout
- kStatus_FLEXSPI_IpCommandSequenceError – IP command sequence error detected
- kStatus_FLEXSPI_IpCommandGrantTimeout – IP command grant timeout detected

status_t FLEXSPI_ReadBlocking(FLEXSPI_Type *base, uint8_t *buffer, size_t size)

Receives a buffer of data bytes using a blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – FLEXSPI peripheral base address
- buffer – The data bytes to send
- size – The number of data bytes to receive

Return values

- kStatus_Success – read success without error
- kStatus_FLEXSPI_SequenceExecutionTimeout – sequence execution timeout
- kStatus_FLEXSPI_IpCommandSequenceError – IP command sequence error detected
- kStatus_FLEXSPI_IpCommandGrantTimeout – IP command grant timeout detected

status_t FLEXSPI_TransferBlocking(FLEXSPI_Type *base, *flexspi_transfer_t* *xfer)

Execute command to transfer a buffer data bytes using a blocking method.

Parameters

- base – FLEXSPI peripheral base address
- xfer – pointer to the transfer structure.

Return values

- kStatus_Success – command transfer success without error
- kStatus_FLEXSPI_SequenceExecutionTimeout – sequence execution timeout
- kStatus_FLEXSPI_IpCommandSequenceError – IP command sequence error detected
- kStatus_FLEXSPI_IpCommandGrantTimeout – IP command grant timeout detected

void FLEXSPI_TransferCreateHandle(FLEXSPI_Type *base, *flexspi_handle_t* *handle,
flexspi_transfer_callback_t callback, void *userData)

Initializes the FLEXSPI handle which is used in transactional functions.

Parameters

- base – FLEXSPI peripheral base address.
- handle – pointer to *flexspi_handle_t* structure to store the transfer state.
- callback – pointer to user callback function.
- userData – user parameter passed to the callback function.

status_t FLEXSPI_TransferNonBlocking(FLEXSPI_Type *base, *flexspi_handle_t* *handle,
flexspi_transfer_t *xfer)

Performs a interrupt non-blocking transfer on the FLEXSPI bus.

Note: Calling the API returns immediately after transfer initiates. The user needs to call `FLEXSPI_GetTransferCount` to poll the transfer status to check whether the transfer is finished. If the return status is not `kStatus_FLEXSPI_Busy`, the transfer is finished. For `FLEXSPI_Read`, the `dataSize` should be multiple of rx watermark level, or FLEXSPI could not read data properly.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure which stores the transfer state.
- `xfer` – pointer to `flexspi_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_FLEXSPI_Busy` – Previous transmission still not finished.

`status_t` `FLEXSPI_TransferGetCount(FLEXSPI_Type *base, flexspi_handle_t *handle, size_t *count)`

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – count is Invalid.
- `kStatus_Success` – Successfully return the count.

`void` `FLEXSPI_TransferAbort(FLEXSPI_Type *base, flexspi_handle_t *handle)`

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure which stores the transfer state

`void` `FLEXSPI_TransferHandleIRQ(FLEXSPI_Type *base, flexspi_handle_t *handle)`

Master interrupt handler.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure.

`FSL_FLEXSPI_DRIVER_VERSION`

FLEXSPI driver version.

Status structure of FLEXSPI.

Values:

enumerator kStatus_FLEXSPI_Busy

FLEXSPI is busy

enumerator kStatus_FLEXSPI_SequenceExecutionTimeout

Sequence execution timeout error occurred during FLEXSPI transfer.

enumerator kStatus_FLEXSPI_IpCommandSequenceError

IP command Sequence execution timeout error occurred during FLEXSPI transfer.

enumerator kStatus_FLEXSPI_IpCommandGrantTimeout

IP command grant timeout error occurred during FLEXSPI transfer.

CMD definition of FLEXSPI, use to form LUT instruction, `_flexspi_command`.

Values:

enumerator kFLEXSPI_Command_STOP

Stop execution, deassert CS.

enumerator kFLEXSPI_Command_SDR

Transmit Command code to Flash, using SDR mode.

enumerator kFLEXSPI_Command_RADDR_SDR

Transmit Row Address to Flash, using SDR mode.

enumerator kFLEXSPI_Command_CADDR_SDR

Transmit Column Address to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE1_SDR

Transmit 1-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE2_SDR

Transmit 2-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE4_SDR

Transmit 4-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE8_SDR

Transmit 8-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_WRITE_SDR

Transmit Programming Data to Flash, using SDR mode.

enumerator kFLEXSPI_Command_READ_SDR

Receive Read Data from Flash, using SDR mode.

enumerator kFLEXSPI_Command_LEARN_SDR

Receive Read Data or Preamble bit from Flash, SDR mode.

enumerator kFLEXSPI_Command_DATSZ_SDR

Transmit Read/Program Data size (byte) to Flash, SDR mode.

enumerator kFLEXSPI_Command_DUMMY_SDR

Leave data lines undriven by FlexSPI controller.

enumerator kFLEXSPI_Command_DUMMY_RWDS_SDR

Leave data lines undriven by FlexSPI controller, dummy cycles decided by RWDS.

enumerator kFLEXSPI_Command_DDR

Transmit Command code to Flash, using DDR mode.

enumerator kFLEXSPI_Command_RADDR_DDR

Transmit Row Address to Flash, using DDR mode.

enumerator kFLEXSPI_Command_CADDR_DDR

Transmit Column Address to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE1_DDR

Transmit 1-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE2_DDR

Transmit 2-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE4_DDR

Transmit 4-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE8_DDR

Transmit 8-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_WRITE_DDR

Transmit Programming Data to Flash, using DDR mode.

enumerator kFLEXSPI_Command_READ_DDR

Receive Read Data from Flash, using DDR mode.

enumerator kFLEXSPI_Command_LEARN_DDR

Receive Read Data or Preamble bit from Flash, DDR mode.

enumerator kFLEXSPI_Command_DATSZ_DDR

Transmit Read/Program Data size (byte) to Flash, DDR mode.

enumerator kFLEXSPI_Command_DUMMY_DDR

Leave data lines undriven by FlexSPI controller.

enumerator kFLEXSPI_Command_DUMMY_RWDS_DDR

Leave data lines undriven by FlexSPI controller, dummy cycles decided by RWDS.

enumerator kFLEXSPI_Command_JUMP_ON_CS

Stop execution, deassert CS and save operand[7:0] as the instruction start pointer for next sequence

enum _flexspi_pad

pad definition of FLEXSPI, use to form LUT instruction.

Values:

enumerator kFLEXSPI_1PAD

Transmit command/address and transmit/receive data only through DATA0/DATA1.

enumerator kFLEXSPI_2PAD

Transmit command/address and transmit/receive data only through DATA[1:0].

enumerator kFLEXSPI_4PAD

Transmit command/address and transmit/receive data only through DATA[3:0].

enumerator kFLEXSPI_8PAD

Transmit command/address and transmit/receive data only through DATA[7:0].

enum _flexspi_flags

FLEXSPI interrupt status flags.

Values:

enumerator kFLEXSPI_SequenceExecutionTimeoutFlag
Sequence execution timeout.

enumerator kFLEXSPI_AhbBusErrorFlag
AHB Bus error flag.

enumerator kFLEXSPI_SckStoppedBecauseTxEmptyFlag
SCK is stopped during command sequence because Async TX FIFO empty.

enumerator kFLEXSPI_SckStoppedBecauseRxFullFlag
SCK is stopped during command sequence because Async RX FIFO full.

enumerator kFLEXSPI_IpTxFifoWatermarkEmptyFlag
IP TX FIFO WaterMark empty.

enumerator kFLEXSPI_IpRxFifoWatermarkAvailableFlag
IP RX FIFO WaterMark available.

enumerator kFLEXSPI_AhbCommandSequenceErrorFlag
AHB triggered Command Sequences Error.

enumerator kFLEXSPI_IpCommandSequenceErrorFlag
IP triggered Command Sequences Error.

enumerator kFLEXSPI_AhbCommandGrantTimeoutFlag
AHB triggered Command Sequences Grant Timeout.

enumerator kFLEXSPI_IpCommandGrantTimeoutFlag
IP triggered Command Sequences Grant Timeout.

enumerator kFLEXSPI_IpCommandExecutionDoneFlag
IP triggered Command Sequences Execution finished.

enumerator kFLEXSPI_AllInterruptFlags
All flags.

enum _flexspi_read_sample_clock
FLEXSPI sample clock source selection for Flash Reading.

Values:

enumerator kFLEXSPI_ReadSampleClkLoopbackInternally
Dummy Read strobe generated by FlexSPI Controller and loopback internally.

enumerator kFLEXSPI_ReadSampleClkLoopbackFromDqsPad
Dummy Read strobe generated by FlexSPI Controller and loopback from DQS pad.

enumerator kFLEXSPI_ReadSampleClkLoopbackFromSckPad
SCK output clock and loopback from SCK pad.

enumerator kFLEXSPI_ReadSampleClkExternalInputFromDqsPad
Flash provided Read strobe and input from DQS pad.

enum _flexspi_cs_interval_cycle_unit
FLEXSPI interval unit for flash device select.

Values:

enumerator kFLEXSPI_CsIntervalUnit1SckCycle
Chip selection interval: CSINTERVAL * 1 serial clock cycle.

enumerator kFLEXSPI_CsIntervalUnit256SckCycle
Chip selection interval: CSINTERVAL * 256 serial clock cycle.

enum _flexspi_ahb_write_wait_unit

FLEXSPI AHB wait interval unit for writing.

Values:

enumerator kFLEXSPI_AhbWriteWaitUnit2AhbCycle

AWRWAIT unit is 2 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit8AhbCycle

AWRWAIT unit is 8 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit32AhbCycle

AWRWAIT unit is 32 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit128AhbCycle

AWRWAIT unit is 128 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit512AhbCycle

AWRWAIT unit is 512 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit2048AhbCycle

AWRWAIT unit is 2048 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit8192AhbCycle

AWRWAIT unit is 8192 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit32768AhbCycle

AWRWAIT unit is 32768 ahb clock cycle.

enum _flexspi_ip_error_code

Error Code when IP command Error detected.

Values:

enumerator kFLEXSPI_IpCmdErrorNoError

No error.

enumerator kFLEXSPI_IpCmdErrorJumpOnCsInIpCmd

IP command with JMP_ON_CS instruction used.

enumerator kFLEXSPI_IpCmdErrorUnknownOpCode

Unknown instruction opcode in the sequence.

enumerator kFLEXSPI_IpCmdErrorSdrDummyInDdrSequence

Instruction DUMMY_SDR/DUMMY_RWDS_SDR used in DDR sequence.

enumerator kFLEXSPI_IpCmdErrorDdrDummyInSdrSequence

Instruction DUMMY_DDR/DUMMY_RWDS_DDR used in SDR sequence.

enumerator kFLEXSPI_IpCmdErrorInvalidAddress

Flash access start address exceed the whole flash address range (A1/A2/B1/B2).

enumerator kFLEXSPI_IpCmdErrorSequenceExecutionTimeout

Sequence execution timeout.

enumerator kFLEXSPI_IpCmdErrorFlashBoundaryAcrosss

Flash boundary crossed.

enum _flexspi_ahb_error_code

Error Code when AHB command Error detected.

Values:

enumerator kFLEXSPI_AhbCmdErrorNoError

No error.

enumerator kFLEXSPI_AhbCmdErrorJumpOnCsInWriteCmd

AHB Write command with JMP_ON_CS instruction used in the sequence.

enumerator kFLEXSPI_AhbCmdErrorUnknownOpCode

Unknown instruction opcode in the sequence.

enumerator kFLEXSPI_AhbCmdErrorSdrDummyInDdrSequence

Instruction DUMMY_SDR/DUMMY_RWDS_SDR used in DDR sequence.

enumerator kFLEXSPI_AhbCmdErrorDdrDummyInSdrSequence

Instruction DUMMY_DDR/DUMMY_RWDS_DDR used in SDR sequence.

enumerator kFLEXSPI_AhbCmdSequenceExecutionTimeout

Sequence execution timeout.

enum _flexspi_port

FLEXSPI operation port select.

Values:

enumerator kFLEXSPI_PortA1

Access flash on A1 port.

enumerator kFLEXSPI_PortA2

Access flash on A2 port.

enumerator kFLEXSPI_PortB1

Access flash on B1 port.

enumerator kFLEXSPI_PortB2

Access flash on B2 port.

enumerator kFLEXSPI_PortCount

enum _flexspi_arb_command_source

Trigger source of current command sequence granted by arbitrator.

Values:

enumerator kFLEXSPI_AhbReadCommand

enumerator kFLEXSPI_AhbWriteCommand

enumerator kFLEXSPI_IpCommand

enumerator kFLEXSPI_SuspendedCommand

enum _flexspi_command_type

Command type.

Values:

enumerator kFLEXSPI_Command

FlexSPI operation: Only command, both TX and Rx buffer are ignored.

enumerator kFLEXSPI_Config

FlexSPI operation: Configure device mode, the TX fifo size is fixed in LUT.

enumerator kFLEXSPI_Read

enumerator kFLEXSPI_Write

```

typedef enum _flexspi_pad flexspi_pad_t
    pad definition of FLEXSPI, use to form LUT instruction.
typedef enum _flexspi_flags flexspi_flags_t
    FLEXSPI interrupt status flags.
typedef enum _flexspi_read_sample_clock flexspi_read_sample_clock_t
    FLEXSPI sample clock source selection for Flash Reading.
typedef enum _flexspi_cs_interval_cycle_unit flexspi_cs_interval_cycle_unit_t
    FLEXSPI interval unit for flash device select.
typedef enum _flexspi_ahb_write_wait_unit flexspi_ahb_write_wait_unit_t
    FLEXSPI AHB wait interval unit for writing.
typedef enum _flexspi_ip_error_code flexspi_ip_error_code_t
    Error Code when IP command Error detected.
typedef enum _flexspi_ahb_error_code flexspi_ahb_error_code_t
    Error Code when AHB command Error detected.
typedef enum _flexspi_port flexspi_port_t
    FLEXSPI operation port select.
typedef enum _flexspi_arb_command_source flexspi_arb_command_source_t
    Trigger source of current command sequence granted by arbitrator.
typedef enum _flexspi_command_type flexspi_command_type_t
    Command type.
typedef struct _flexspi_ahbBuffer_config flexspi_ahbBuffer_config_t
typedef struct _flexspi_ahbBuffers_ctrl flexspi_ahbBuffers_ctrl_t
    Structure to control all AHB buffers.
typedef struct _flexspi_config flexspi_config_t
    FLEXSPI configuration structure.
typedef struct _flexspi_device_config flexspi_device_config_t
    External device configuration items.
typedef struct _flexspi_transfer flexspi_transfer_t
    Transfer structure for FLEXSPI.
typedef struct _flexspi_handle flexspi_handle_t
typedef void (*flexspi_transfer_callback_t)(FLEXSPI_Type *base, flexspi_handle_t *handle,
status_t status, void *userData)
    FLEXSPI transfer callback function.
typedef struct _flexspi_addr_map_config flexspi_addr_map_config_t
    Address mapping configuration structure.
FSL_FEATURE_FLEXSPI_AHB_BUFFER_COUNT
FLEXSPI_LUT_SEQ(cmd0, pad0, op0, cmd1, pad1, op1)
    Formula to form FLEXSPI instructions in LUT table.
struct _flexspi_ahbBuffer_config
    #include <fsl_flexspi.h>

```

Public Members

uint8_t priority

This priority for AHB Master Read which this AHB RX Buffer is assigned.

uint8_t masterIndex

AHB Master ID the AHB RX Buffer is assigned.

uint16_t bufferSize

AHB buffer size in byte.

bool enablePrefetch

AHB Read Prefetch Enable for current AHB RX Buffer corresponding Master, allows prefetch disable/enable separately for each master.

struct _flexspi_ahbBuffers_ctrl

#include <fsl_flexspi.h> Structure to control all AHB buffers.

Public Members

flexspi_ahbBuffer_config_t buffer[FSL_FEATURE_FLEXSPI_AHB_BUFFER_COUNTn(0)]

Configurations of all AHB buffers.

struct _flexspi_config

#include <fsl_flexspi.h> FLEXSPI configuration structure.

Public Members

flexspi_read_sample_clock_t rxSampleClock

Sample Clock source selection for Flash Reading.

bool enableSckFreeRunning

Enable/disable SCK output free-running.

bool enableCombination

Enable/disable combining PORT A and B Data Pins (SIOA[3:0] and SIOB[3:0]) to support Flash Octal mode.

bool enableDoze

Enable/disable doze mode support.

bool enableHalfSpeedAccess

Enable/disable divide by 2 of the clock for half speed commands.

bool enableSckBDiffOpt

Enable/disable SCKB pad use as SCKA differential clock output, when enable, Port B flash access is not available.

bool enableSameConfigForAll

Enable/disable same configuration for all connected devices when enabled, same configuration in FLASHA1CRx is applied to all.

uint16_t seqTimeoutCycle

Timeout wait cycle for command sequence execution, timeout after ahbGrantTimeoutCycle*1024 serial root clock cycles.

uint8_t ipGrantTimeoutCycle

Timeout wait cycle for IP command grant, timeout after ipGrantTimeoutCycle*1024 AHB clock cycles.

uint8_t txWatermark
FLEXSPI IP transmit watermark value.

uint8_t rxWatermark
FLEXSPI receive watermark value.

struct __flexspi_device_config
#include <fsl_flexspi.h> External device configuration items.

Public Members

uint32_t flexspiRootClk
FLEXSPI serial root clock.

bool isSck2Enabled
FLEXSPI use SCK2.

uint32_t flashSize
Flash size in KByte.

bool addressShift
Address shift.

flexspi_cs_interval_cycle_unit_t CSIntervalUnit
CS interval unit, 1 or 256 cycle.

uint16_t CSInterval
CS line assert interval, multiply CS interval unit to get the CS line assert interval cycles.

uint8_t CSHoldTime
CS line hold time.

uint8_t CSSetupTime
CS line setup time.

uint8_t dataValidTime
Data valid time for external device.

uint8_t columnSpace
Column space size.

bool enableWordAddress
If enable word address.

uint8_t AWRSeqIndex
Sequence ID for AHB write command.

uint8_t AWRSeqNumber
Sequence number for AHB write command.

uint8_t ARDSeqIndex
Sequence ID for AHB read command.

uint8_t ARDSeqNumber
Sequence number for AHB read command.

flexspi_ahb_write_wait_unit_t AHBWriteWaitUnit
AHB write wait unit.

uint16_t AHBWriteWaitInterval
AHB write wait interval, multiply AHB write interval unit to get the AHB write wait cycles.

bool enableWriteMask

Enable/Disable FLEXSPI drive DQS pin as write mask when writing to external device.

struct __flexspi_transfer

#include <fsl_flexspi.h> Transfer structure for FLEXSPI.

Public Members

uint32_t deviceAddress

Operation device address.

flexspi_port_t port

Operation port.

flexspi_command_type_t cmdType

Execution command type.

uint8_t seqIndex

Sequence ID for command.

uint8_t SeqNumber

Sequence number for command.

uint32_t *data

Data buffer.

size_t dataSize

Data size in bytes.

struct __flexspi_handle

#include <fsl_flexspi.h> Transfer handle structure for FLEXSPI.

Public Members

uint32_t state

Internal state for FLEXSPI transfer

uint8_t *data

Data buffer.

size_t dataSize

Remaining Data size in bytes.

size_t transferTotalSize

Total Data size in bytes.

flexspi_transfer_callback_t completionCallback

Callback for users while transfer finish or error occurred

void *userData

FLEXSPI callback function parameter.

struct __flexspi_addr_map_config

#include <fsl_flexspi.h> Address mapping configuration structure.

Public Members

uint32_t addrStart
Remapping start address.

uint32_t addrEnd
Remapping end address.

uint32_t addrOffset
Address offset.

bool remapEnable
Enable address remapping.

struct ahbConfig

Public Members

uint8_t ahbGrantTimeoutCycle
Timeout wait cycle for AHB command grant, timeout after ahbGrantTimeoutCycle*1024 AHB clock cycles.

uint16_t ahbBusTimeoutCycle
Timeout wait cycle for AHB read/write access, timeout after ahbBusTimeoutCycle*1024 AHB clock cycles.

uint8_t resumeWaitCycle
Wait cycle for idle state before suspended command sequence resume, timeout after ahbBusTimeoutCycle AHB clock cycles.

flexspi_ahbBuffer_config_t buffer[FSL_FEATURE_FLEXSPI_AHB_BUFFER_COUNTn(0)]
AHB buffer size.

bool enableClearAHBBufferOpt
Enable/disable automatically clean AHB RX Buffer and TX Buffer when FLEXSPI returns STOP mode ACK.

bool enableReadAddressOpt
Enable/disable remove AHB read burst start address alignment limitation. when enable, there is no AHB read burst start address alignment limitation.

bool enableAHBPrefetch
Enable/disable AHB read prefetch feature, when enabled, FLEXSPI will fetch more data than current AHB burst.

bool enableAHBBufferable
Enable/disable AHB bufferable write access support, when enabled, FLEXSPI return before waiting for command execution finished.

bool enableAHBCachable
Enable AHB bus cachable read access support.

2.26 FLEXSPI eDMA Driver

```
void FLEXSPI_TransferCreateHandleEDMA(FLEXSPI_Type *base, flexspi_edma_handle_t  
                                     *handle, flexspi_edma_callback_t callback, void  
                                     *userData, edma_handle_t *txDmaHandle,  
                                     edma_handle_t *rxDmaHandle)
```

Initializes the FLEXSPI handle for transfer which is used in transactional functions and set the callback.

Parameters

- base – FLEXSPI peripheral base address
- handle – Pointer to flexspi_edma_handle_t structure
- callback – FLEXSPI callback, NULL means no callback.
- userData – User callback function data.
- txDmaHandle – User requested DMA handle for TX DMA transfer.
- rxDmaHandle – User requested DMA handle for RX DMA transfer.

```
void FLEXSPI_TransferUpdateSizeEDMA(FLEXSPI_Type *base, flexspi_edma_handle_t *handle,  
                                     flexspi_edma_transfer_nsize_t nsize)
```

Update FLEXSPI EDMA transfer source data transfer size(SSIZE) and destination data transfer size(DSIZE).

See also:

flexspi_edma_transfer_nsize_t .

Parameters

- base – FLEXSPI peripheral base address
- handle – Pointer to flexspi_edma_handle_t structure
- nsize – FLEXSPI DMA transfer data transfer size(SSIZE/DSIZE), by default the size is kFLEXSPI_EDMAAnSize1Bytes(one byte).

```
status_t FLEXSPI_TransferEDMA(FLEXSPI_Type *base, flexspi_edma_handle_t *handle,  
                              flexspi_transfer_t *xfer)
```

Transfers FLEXSPI data using an eDMA non-blocking method.

This function writes/receives data to/from the FLEXSPI transmit/receive FIFO. This function is non-blocking.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_edma_handle_t structure
- xfer – FLEXSPI transfer structure.

Return values

- kStatus_FLEXSPI_Busy – FLEXSPI is busy transfer.
- kStatus_InvalidArgument – The watermark configuration is invalid, the watermark should be power of 2 to do successfully EDMA transfer.
- kStatus_Success – FLEXSPI successfully start edma transfer.

```
void FLEXSPI_TransferAbortEDMA(FLEXSPI_Type *base, flexspi_edma_handle_t *handle)
```

Aborts the transfer data using eDMA.

This function aborts the transfer data using eDMA.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_edma_handle_t structure

```
status_t FLEXSPI_TransferGetTransferCountEDMA(FLEXSPI_Type *base, flexspi_edma_handle_t
                                             *handle, size_t *count)
```

Gets the transferred counts of transfer.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_edma_handle_t structure.
- count – Bytes transfer.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

```
FSL_FLEXSPI_EDMA_DRIVER_VERSION
```

FLEXSPI EDMA driver version.

```
enum _flexspi_edma_ntransfer_size
eDMA transfer configuration
```

Values:

```
enumerator kFLEXPSI_EDMAAnSize1Bytes
```

Source/Destination data transfer size is 1 byte every time

```
enumerator kFLEXPSI_EDMAAnSize2Bytes
```

Source/Destination data transfer size is 2 bytes every time

```
enumerator kFLEXPSI_EDMAAnSize4Bytes
```

Source/Destination data transfer size is 4 bytes every time

```
enumerator kFLEXPSI_EDMAAnSize8Bytes
```

Source/Destination data transfer size is 8 bytes every time

```
enumerator kFLEXPSI_EDMAAnSize32Bytes
```

Source/Destination data transfer size is 32 bytes every time

```
typedef struct _flexspi_edma_handle flexspi_edma_handle_t
```

```
typedef void (*flexspi_edma_callback_t)(FLEXSPI_Type *base, flexspi_edma_handle_t *handle,
status_t status, void *userData)
```

FLEXSPI eDMA transfer callback function for finish and error.

```
typedef enum _flexspi_edma_ntransfer_size flexspi_edma_transfer_nsize_t
eDMA transfer configuration
```

```
struct _flexspi_edma_handle
```

#include <fsl_flexspi_edma.h> FLEXSPI DMA transfer handle, users should not touch the content of the handle.

Public Members

```
edma_handle_t *txDmaHandle
```

eDMA handler for FLEXSPI Tx.

```
edma_handle_t *rxDmaHandle
```

eDMA handler for FLEXSPI Rx.

`size_t` transferSize
Bytes need to transfer.

`flexspi_edma_transfer_nsize_t` nsize
eDMA SSIZE/DSIZE in each transfer.

`uint8_t` nbytes
eDMA minor byte transfer count initially configured.

`uint8_t` count
The transfer data count in a DMA request.

`uint32_t` state
Internal state for FLEXSPI eDMA transfer.

`flexspi_edma_callback_t` completionCallback
A callback function called after the eDMA transfer is finished.

`void *userData`
User callback parameter

2.27 GPC: General Power Controller Driver

`FSL_GPC_DRIVER_VERSION`

GPC driver version 2.1.1.

`static inline void GPC_AllowIRQs(GPC_Type *base)`
Allow all the IRQ/Events within the charge of GPC.

Parameters

- base – GPC peripheral base address.

`static inline void GPC_DisallowIRQs(GPC_Type *base)`
Disallow all the IRQ/Events within the charge of GPC.

Parameters

- base – GPC peripheral base address.

`void GPC_EnableIRQ(GPC_Type *base, uint32_t irqId)`
Enable the IRQ.

Parameters

- base – GPC peripheral base address.
- irqId – ID number of IRQ to be enabled, available range is 32-159. 0-31 is available in some platforms.

`void GPC_DisableIRQ(GPC_Type *base, uint32_t irqId)`
Disable the IRQ.

Parameters

- base – GPC peripheral base address.
- irqId – ID number of IRQ to be disabled, available range is 32-159. 0-31 is available in some platforms.

```
bool GPC_GetIRQStatusFlag(GPC_Type *base, uint32_t irqId)
```

Get the IRQ/Event flag.

Parameters

- base – GPC peripheral base address.
- irqId – ID number of IRQ to be enabled, available range is 32-159. 0-31 is available in some platforms.

Returns

Indicated IRQ/Event is asserted or not.

```
static inline void GPC_RequestL2CachePowerDown(GPC_Type *base, bool enable)
```

L2 Cache Power Gate Enable.

This function configures the L2 cache if it will keep power when in low power mode. When the L2 cache power is OFF, L2 cache will be power down once when CPU core is power down and will be hardware invalidated automatically when CPU core is re-power up. When the L2 cache power is ON, L2 cache will keep power on even if CPU core is power down and will not be hardware invalidated. When CPU core is re-power up, the default setting is OFF.

Parameters

- base – GPC peripheral base address.
- enable – Enable the request or not.

```
static inline void GPC_RequestPdram0PowerDown(GPC_Type *base, bool enable)
```

FLEXRAM PDRAM0 Power Gate Enable.

This function configures the FLEXRAM PDRAM0 if it will keep power when cpu core is power down. When the PDRAM0 Power is 1, PDRAM0 will be power down once when CPU core is power down. When the PDRAM0 Power is 0, PDRAM0 will keep power on even if CPU core is power down. When CPU core is re-power up, the default setting is 1.

Parameters

- base – GPC peripheral base address.
- enable – Enable the request or not.

```
static inline void GPC_RequestVADCPowerDown(GPC_Type *base, bool enable)
```

VADC power down.

This function requests the VADC power down.

Parameters

- base – GPC peripheral base address.
- enable – Enable the request or not.

```
static inline bool GPC_GetVADCPowerDownFlag(GPC_Type *base)
```

Checks if the VADC is power off.

Parameters

- base – GPC peripheral base address.

Returns

Whether the VADC is power off or not.

```
static inline bool GPC_HasDVFS0ChangeRequest(GPC_Type *base)
```

Checks if the DVFS0 is requesting for frequency/voltage update.

Parameters

- base – GPC peripheral base address.

Returns

Whether the DVFS0 is requesting for frequency/voltage update.

static inline void GPC_RequestDisplayPowerOn(GPC_Type *base, bool enable)

Requests the display power switch sequence.

Parameters

- base – GPC peripheral base address.
- enable – Enable the power on sequence, or the power down sequence.

static inline void GPC_RequestMEGAPowerOn(GPC_Type *base, bool enable)

Requests the MEGA power switch sequence.

Parameters

- base – GPC peripheral base address.
- enable – Enable the power on sequence, or the power down sequence.

2.28 GPIO: General-Purpose Input/Output Driver

void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const *gpio_pin_config_t* *Config)

Initializes the GPIO peripheral according to the specified parameters in the initConfig.

Parameters

- base – GPIO base pointer.
- pin – Specifies the pin number
- Config – pointer to a *gpio_pin_config_t* structure that contains the configuration information.

void GPIO_PinWrite(GPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the individual GPIO pin to logic 1 or 0.

Parameters

- base – GPIO base pointer.
- pin – GPIO port pin number.
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void GPIO_WritePinOutput(GPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the individual GPIO pin to logic 1 or 0.

Deprecated:

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

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- base – GPIO peripheral base pointer (GPIO1, GPIO2, GPIO3, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_SetPinsOutput(GPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple GPIO pins to the logic 1.

Deprecated:

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

```
static inline void GPIO_PortClear(GPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer (GPIO1, GPIO2, GPIO3, and so on.)
- mask – GPIO pin number macro

```
static inline void GPIO_ClearPinsOutput(GPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple GPIO pins to the logic 0.

Deprecated:

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

```
static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t mask)
```

Reverses the current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer (GPIO1, GPIO2, GPIO3, and so on.)
- mask – GPIO pin number macro

```
static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t pin)
```

Reads the current input value of the GPIO port.

Parameters

- base – GPIO base pointer.
- pin – GPIO port pin number.

Return values

GPIO – port input value.

```
static inline uint32_t GPIO_ReadPinInput(GPIO_Type *base, uint32_t pin)
```

Reads the current input value of the GPIO port.

Deprecated:

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

```
static inline uint8_t GPIO_PinReadPadStatus(GPIO_Type *base, uint32_t pin)
```

Reads the current GPIO pin pad status.

Parameters

- base – GPIO base pointer.
- pin – GPIO port pin number.

Return values

GPIO – pin pad status value.

```
static inline uint8_t GPIO_ReadPadStatus(GPIO_Type *base, uint32_t pin)
```

Reads the current GPIO pin pad status.

Deprecated:

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

```
void GPIO_PinSetInterruptConfig(GPIO_Type *base, uint32_t pin, gpio_interrupt_mode_t  
                                pinInterruptMode)
```

Sets the current pin interrupt mode.

Parameters

- base – GPIO base pointer.
- pin – GPIO port pin number.
- pinInterruptMode – pointer to a gpio_interrupt_mode_t structure that contains the interrupt mode information.

```
static inline void GPIO_SetPinInterruptConfig(GPIO_Type *base, uint32_t pin,  
                                              gpio_interrupt_mode_t pinInterruptMode)
```

Sets the current pin interrupt mode.

Deprecated:

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

```
static inline void GPIO_PortEnableInterrupts(GPIO_Type *base, uint32_t mask)
```

Enables the specific pin interrupt.

Parameters

- base – GPIO base pointer.
- mask – GPIO pin number macro.

```
static inline void GPIO_EnableInterrupts(GPIO_Type *base, uint32_t mask)
```

Enables the specific pin interrupt.

Parameters

- base – GPIO base pointer.
- mask – GPIO pin number macro.

```
static inline void GPIO_PortDisableInterrupts(GPIO_Type *base, uint32_t mask)
```

Disables the specific pin interrupt.

Parameters

- base – GPIO base pointer.
- mask – GPIO pin number macro.

```
static inline void GPIO_DisableInterrupts(GPIO_Type *base, uint32_t mask)
```

Disables the specific pin interrupt.

Deprecated:

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


```
static inline uint32_t GPIO_PortGetInterruptFlags(GPIO_Type *base)
```

Reads individual pin interrupt status.

Parameters

- base – GPIO base pointer.

Return values

current – pin interrupt status flag.

```
static inline uint32_t GPIO_GetPinsInterruptFlags(GPIO_Type *base)
```

Reads individual pin interrupt status.

Parameters

- base – GPIO base pointer.

Return values

current – pin interrupt status flag.

```
static inline void GPIO_PortClearInterruptFlags(GPIO_Type *base, uint32_t mask)
```

Clears pin interrupt flag. Status flags are cleared by writing a 1 to the corresponding bit position.

Parameters

- base – GPIO base pointer.
- mask – GPIO pin number macro.

```
static inline void GPIO_ClearPinsInterruptFlags(GPIO_Type *base, uint32_t mask)
```

Clears pin interrupt flag. Status flags are cleared by writing a 1 to the corresponding bit position.

Parameters

- base – GPIO base pointer.
- mask – GPIO pin number macro.

```
FSL_GPIO_DRIVER_VERSION
```

GPIO driver version.

```
enum _gpio_pin_direction
```

GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput

Set current pin as digital input.

enumerator kGPIO_DigitalOutput

Set current pin as digital output.

```
enum _gpio_interrupt_mode
```

GPIO interrupt mode definition.

Values:

enumerator kGPIO_NoIntmode

Set current pin general IO functionality.

enumerator kGPIO_IntLowLevel

Set current pin interrupt is low-level sensitive.

enumerator kGPIO_IntHighLevel

Set current pin interrupt is high-level sensitive.

enumerator kGPIO_IntRisingEdge

Set current pin interrupt is rising-edge sensitive.

enumerator kGPIO_IntFallingEdge

Set current pin interrupt is falling-edge sensitive.

enumerator kGPIO_IntRisingOrFallingEdge

Enable the edge select bit to override the ICR register's configuration.

typedef enum *_gpio_pin_direction* gpio_pin_direction_t

GPIO direction definition.

typedef enum *_gpio_interrupt_mode* gpio_interrupt_mode_t

GPIO interrupt mode definition.

typedef struct *_gpio_pin_config* gpio_pin_config_t

GPIO Init structure definition.

struct *_gpio_pin_config*

#include <fsl_gpio.h> GPIO Init structure definition.

Public Members

gpio_pin_direction_t direction

Specifies the pin direction.

uint8_t outputLogic

Set a default output logic, which has no use in input

gpio_interrupt_mode_t interruptMode

Specifies the pin interrupt mode, a value of *gpio_interrupt_mode_t*.

2.29 GPT: General Purpose Timer

void GPT_Init(GPT_Type *base, const *gpt_config_t* *initConfig)

Initialize GPT to reset state and initialize running mode.

Parameters

- base – GPT peripheral base address.
- initConfig – GPT mode setting configuration.

void GPT_Deinit(GPT_Type *base)

Disables the module and gates the GPT clock.

Parameters

- base – GPT peripheral base address.

void GPT_GetDefaultConfig(*gpt_config_t* *config)

Fills in the GPT configuration structure with default settings.

The default values are:

```
config->clockSource = kGPT_ClockSource_Periph;
config->divider = 1U;
config->enableRunInStop = true;
config->enableRunInWait = true;
config->enableRunInDoze = false;
```

(continues on next page)

(continued from previous page)

```
config->enableRunInDbg = false;
config->enableFreeRun = false;
config->enableMode = true;
```

Parameters

- config – Pointer to the user configuration structure.

```
static inline void GPT_SoftwareReset(GPT_Type *base)
```

Software reset of GPT module.

Parameters

- base – GPT peripheral base address.

```
static inline void GPT_SetClockSource(GPT_Type *base, gpt_clock_source_t gptClkSource)
```

Set clock source of GPT.

Parameters

- base – GPT peripheral base address.
- gptClkSource – Clock source (see `gpt_clock_source_t` typedef enumeration).

```
static inline gpt_clock_source_t GPT_GetClockSource(GPT_Type *base)
```

Get clock source of GPT.

Parameters

- base – GPT peripheral base address.

Returns

clock source (see `gpt_clock_source_t` typedef enumeration).

```
static inline void GPT_SetClockDivider(GPT_Type *base, uint32_t divider)
```

Set pre scaler of GPT.

Parameters

- base – GPT peripheral base address.
- divider – Divider of GPT (1-4096).

```
static inline uint32_t GPT_GetClockDivider(GPT_Type *base)
```

Get clock divider in GPT module.

Parameters

- base – GPT peripheral base address.

Returns

clock divider in GPT module (1-4096).

```
static inline void GPT_SetOscClockDivider(GPT_Type *base, uint32_t divider)
```

OSC 24M pre-scaler before selected by clock source.

Parameters

- base – GPT peripheral base address.
- divider – OSC Divider(1-16).

```
static inline uint32_t GPT_GetOscClockDivider(GPT_Type *base)
```

Get OSC 24M clock divider in GPT module.

Parameters

- base – GPT peripheral base address.

Returns

OSC clock divider in GPT module (1-16).

static inline void GPT_StartTimer(GPT_Type *base)

Start GPT timer.

Parameters

- base – GPT peripheral base address.

static inline void GPT_StopTimer(GPT_Type *base)

Stop GPT timer.

Parameters

- base – GPT peripheral base address.

static inline uint32_t GPT_GetCurrentTimerCount(GPT_Type *base)

Reads the current GPT counting value.

Parameters

- base – GPT peripheral base address.

Returns

Current GPT counter value.

static inline void GPT_SetInputOperationMode(GPT_Type *base, *gpt_input_capture_channel_t* channel, *gpt_input_operation_mode_t* mode)

Set GPT operation mode of input capture channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT capture channel (see *gpt_input_capture_channel_t* typedef enumeration).
- mode – GPT input capture operation mode (see *gpt_input_operation_mode_t* typedef enumeration).

static inline *gpt_input_operation_mode_t* GPT_GetInputOperationMode(GPT_Type *base, *gpt_input_capture_channel_t* channel)

Get GPT operation mode of input capture channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT capture channel (see *gpt_input_capture_channel_t* typedef enumeration).

Returns

GPT input capture operation mode (see *gpt_input_operation_mode_t* typedef enumeration).

static inline uint32_t GPT_GetInputCaptureValue(GPT_Type *base, *gpt_input_capture_channel_t* channel)

Get GPT input capture value of certain channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT capture channel (see *gpt_input_capture_channel_t* typedef enumeration).

Returns

GPT input capture value.

```
static inline void GPT_SetOutputOperationMode(GPT_Type *base,
                                              gpt_output_compare_channel_t channel,
                                              gpt_output_operation_mode_t mode)
```

Set GPT operation mode of output compare channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT output compare channel (see `gpt_output_compare_channel_t` typedef enumeration).
- mode – GPT output operation mode (see `gpt_output_operation_mode_t` typedef enumeration).

```
static inline gpt_output_operation_mode_t GPT_GetOutputOperationMode(GPT_Type *base,
                                                                    gpt_output_compare_channel_t channel)
```

Get GPT operation mode of output compare channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT output compare channel (see `gpt_output_compare_channel_t` typedef enumeration).

Returns

GPT output operation mode (see `gpt_output_operation_mode_t` typedef enumeration).

```
static inline void GPT_SetOutputCompareValue(GPT_Type *base, gpt_output_compare_channel_t channel, uint32_t value)
```

Set GPT output compare value of output compare channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT output compare channel (see `gpt_output_compare_channel_t` typedef enumeration).
- value – GPT output compare value.

```
static inline uint32_t GPT_GetOutputCompareValue(GPT_Type *base,
                                                gpt_output_compare_channel_t channel)
```

Get GPT output compare value of output compare channel.

Parameters

- base – GPT peripheral base address.
- channel – GPT output compare channel (see `gpt_output_compare_channel_t` typedef enumeration).

Returns

GPT output compare value.

```
static inline void GPT_ForceOutput(GPT_Type *base, gpt_output_compare_channel_t channel)
```

Force GPT output action on output compare channel, ignoring comparator.

Parameters

- base – GPT peripheral base address.

- `channel` – GPT output compare channel (see `gpt_output_compare_channel_t` typedef enumeration).

static inline void GPT_EnableInterrupts(GPT_Type *base, uint32_t mask)

Enables the selected GPT interrupts.

Parameters

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

static inline void GPT_DisableInterrupts(GPT_Type *base, uint32_t mask)

Disables the selected GPT interrupts.

Parameters

- `base` – GPT peripheral base address
- `mask` – The interrupts to disable. This is a logical OR of members of the enumeration `gpt_interrupt_enable_t`

static inline uint32_t GPT_GetEnabledInterrupts(GPT_Type *base)

Gets the enabled GPT interrupts.

Parameters

- `base` – GPT peripheral base address

Returns

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

static inline uint32_t GPT_GetStatusFlags(GPT_Type *base, *gpt_status_flag_t* flags)

Get GPT status flags.

Parameters

- `base` – GPT peripheral base address.
- `flags` – GPT status flag mask (see `gpt_status_flag_t` for bit definition).

Returns

GPT status, each bit represents one status flag.

static inline void GPT_ClearStatusFlags(GPT_Type *base, *gpt_status_flag_t* flags)

Clears the GPT status flags.

Parameters

- `base` – GPT peripheral base address.
- `flags` – GPT status flag mask (see `gpt_status_flag_t` for bit definition).

FSL_GPT_DRIVER_VERSION

enum `_gpt_clock_source`

List of clock sources.

Note: Actual number of clock sources is SoC dependent

Values:

enumerator `kGPT_ClockSource_Off`
GPT Clock Source Off.

enumerator kGPT_ClockSource_Periph

GPT Clock Source from Peripheral Clock.

enumerator kGPT_ClockSource_HighFreq

GPT Clock Source from High Frequency Reference Clock.

enumerator kGPT_ClockSource_Ext

GPT Clock Source from external pin.

enumerator kGPT_ClockSource_LowFreq

GPT Clock Source from Low Frequency Reference Clock.

enumerator kGPT_ClockSource_Osc

GPT Clock Source from Crystal oscillator.

enum _gpt_input_capture_channel

List of input capture channel number.

Values:

enumerator kGPT_InputCapture_Channel1

GPT Input Capture Channel1.

enumerator kGPT_InputCapture_Channel2

GPT Input Capture Channel2.

enum _gpt_input_operation_mode

List of input capture operation mode.

Values:

enumerator kGPT_InputOperation_Disabled

Don't capture.

enumerator kGPT_InputOperation_RiseEdge

Capture on rising edge of input pin.

enumerator kGPT_InputOperation_FallEdge

Capture on falling edge of input pin.

enumerator kGPT_InputOperation_BothEdge

Capture on both edges of input pin.

enum _gpt_output_compare_channel

List of output compare channel number.

Values:

enumerator kGPT_OutputCompare_Channel1

Output Compare Channel1.

enumerator kGPT_OutputCompare_Channel2

Output Compare Channel2.

enumerator kGPT_OutputCompare_Channel3

Output Compare Channel3.

enum _gpt_output_operation_mode

List of output compare operation mode.

Values:

enumerator kGPT_OutputOperation_Disconnected

Don't change output pin.

enumerator kGPT_OutputOperation_Toggle
Toggle output pin.

enumerator kGPT_OutputOperation_Clear
Set output pin low.

enumerator kGPT_OutputOperation_Set
Set output pin high.

enumerator kGPT_OutputOperation_Activelow
Generate a active low pulse on output pin.

enum _gpt_interrupt_enable
List of GPT interrupts.

Values:

enumerator kGPT_OutputCompare1InterruptEnable
Output Compare Channel1 interrupt enable

enumerator kGPT_OutputCompare2InterruptEnable
Output Compare Channel2 interrupt enable

enumerator kGPT_OutputCompare3InterruptEnable
Output Compare Channel3 interrupt enable

enumerator kGPT_InputCapture1InterruptEnable
Input Capture Channel1 interrupt enable

enumerator kGPT_InputCapture2InterruptEnable
Input Capture Channel1 interrupt enable

enumerator kGPT_RollOverFlagInterruptEnable
Counter rolled over interrupt enable

enum _gpt_status_flag
Status flag.

Values:

enumerator kGPT_OutputCompare1Flag
Output compare channel 1 event.

enumerator kGPT_OutputCompare2Flag
Output compare channel 2 event.

enumerator kGPT_OutputCompare3Flag
Output compare channel 3 event.

enumerator kGPT_InputCapture1Flag
Input Capture channel 1 event.

enumerator kGPT_InputCapture2Flag
Input Capture channel 2 event.

enumerator kGPT_RollOverFlag
Counter reaches maximum value and rolled over to 0 event.

typedef enum _gpt_clock_source gpt_clock_source_t
List of clock sources.

Note: Actual number of clock sources is SoC dependent

typedef enum *_gpt_input_capture_channel* gpt_input_capture_channel_t

List of input capture channel number.

typedef enum *_gpt_input_operation_mode* gpt_input_operation_mode_t

List of input capture operation mode.

typedef enum *_gpt_output_compare_channel* gpt_output_compare_channel_t

List of output compare channel number.

typedef enum *_gpt_output_operation_mode* gpt_output_operation_mode_t

List of output compare operation mode.

typedef enum *_gpt_interrupt_enable* gpt_interrupt_enable_t

List of GPT interrupts.

typedef enum *_gpt_status_flag* gpt_status_flag_t

Status flag.

typedef struct *_gpt_init_config* gpt_config_t

Structure to configure the running mode.

struct *_gpt_init_config*

#include <fsl_gpt.h> Structure to configure the running mode.

Public Members

gpt_clock_source_t clockSource

clock source for GPT module.

uint32_t divider

clock divider (prescaler+1) from clock source to counter.

bool enableFreeRun

true: FreeRun mode, false: Restart mode.

bool enableRunInWait

GPT enabled in wait mode.

bool enableRunInStop

GPT enabled in stop mode.

bool enableRunInDoze

GPT enabled in doze mode.

bool enableRunInDbg

GPT enabled in debug mode.

bool enableMode

true: counter reset to 0 when enabled; false: counter retain its value when enabled.

2.30 IOMUXC: IOMUX Controller

enum *_iomuxc_gpr_mode*

Values:

enumerator kIOMUXC_GPR_GlobalInterruptRequest

enumerator kIOMUXC_GPR_SAI1MClkOutputDir

enumerator kIOMUXC_GPR_SAI3MClkOutputDir

enumerator kIOMUXC_GPR_ExcMonitorSlavErrResponse

enumerator kIOMUXC_GPR_AHBClockEnable

enum _iomuxc_gpr_saimclk

Values:

enumerator kIOMUXC_GPR_SAI1MClk1Sel

enumerator kIOMUXC_GPR_SAI1MClk2Sel

enumerator kIOMUXC_GPR_SAI1MClk3Sel

enumerator kIOMUXC_GPR_SAI3MClk3Sel

enum _iomuxc_mqs_pwm_oversample_rate

Values:

enumerator kIOMUXC_MqsPwmOverSampleRate32

enumerator kIOMUXC_MqsPwmOverSampleRate64

typedef enum _iomuxc_gpr_mode iomuxc_gpr_mode_t

typedef enum _iomuxc_gpr_saimclk iomuxc_gpr_saimclk_t

typedef enum _iomuxc_mqs_pwm_oversample_rate iomuxc_mqs_pwm_oversample_rate_t

IOMUXC_GPIO_AD_14_LPI2C1_SCL

IOMUXC_GPIO_AD_14_LPUART3_CTS_B

IOMUXC_GPIO_AD_14_KPP_COL00

IOMUXC_GPIO_AD_14_LPUART4_CTS_B

IOMUXC_GPIO_AD_14_FLEXIO1_IO26

IOMUXC_GPIO_AD_14_GPIOMUX_IO28

IOMUXC_GPIO_AD_14_REF_CLK_24M

IOMUXC_GPIO_AD_14_XBAR1_INOUT02

IOMUXC_GPIO_AD_13_LPI2C1_SDA

IOMUXC_GPIO_AD_13_LPUART3_RTS_B

IOMUXC_GPIO_AD_13_KPP_ROW00

IOMUXC_GPIO_AD_13_LPUART4_RTS_B

IOMUXC_GPIO_AD_13_FLEXIO1_IO25

IOMUXC_GPIO_AD_13_GPIOMUX_IO27

IOMUXC_GPIO_AD_13_ARM_NMI

IOMUXC_GPIO_AD_13_JTAG_TMS

IOMUXC_GPIO_AD_12_LPSPI2_SCK

IOMUXC_GPIO_AD_12_FLEXPWM1_PWM0_X

IOMUXC_GPIO_AD_12_KPP_COL01
IOMUXC_GPIO_AD_12_PIT_TRIGGER01
IOMUXC_GPIO_AD_12_FLEXIO1_IO24
IOMUXC_GPIO_AD_12_GPIOMUX_IO26
IOMUXC_GPIO_AD_12_USB_OTG1_PWR
IOMUXC_GPIO_AD_12_JTAG_TCK
IOMUXC_GPIO_AD_11_LPSPI2_PCS0
IOMUXC_GPIO_AD_11_FLEXPWM1_PWM1_X
IOMUXC_GPIO_AD_11_KPP_ROW01
IOMUXC_GPIO_AD_11_PIT_TRIGGER02
IOMUXC_GPIO_AD_11_FLEXIO1_IO23
IOMUXC_GPIO_AD_11_GPIOMUX_IO25
IOMUXC_GPIO_AD_11_WDOG1_B
IOMUXC_GPIO_AD_11_JTAG_MOD
IOMUXC_GPIO_AD_10_LPSPI2_SDO
IOMUXC_GPIO_AD_10_FLEXPWM1_PWM2_X
IOMUXC_GPIO_AD_10_KPP_COL02
IOMUXC_GPIO_AD_10_PIT_TRIGGER03
IOMUXC_GPIO_AD_10_FLEXIO1_IO22
IOMUXC_GPIO_AD_10_GPIOMUX_IO24
IOMUXC_GPIO_AD_10_USB_OTG1_ID
IOMUXC_GPIO_AD_10_JTAG_TDI
IOMUXC_GPIO_AD_09_LPSPI2_SDI
IOMUXC_GPIO_AD_09_FLEXPWM1_PWM3_X
IOMUXC_GPIO_AD_09_KPP_ROW02
IOMUXC_GPIO_AD_09_ARM_TRACE_SWO
IOMUXC_GPIO_AD_09_FLEXIO1_IO21
IOMUXC_GPIO_AD_09_GPIOMUX_IO23
IOMUXC_GPIO_AD_09_REF_CLK_32K
IOMUXC_GPIO_AD_09_JTAG_TDO
IOMUXC_GPIO_AD_08_LPI2C2_SCL
IOMUXC_GPIO_AD_08_LPUART3_TXD
IOMUXC_GPIO_AD_08_ARM_CM7_TXEV

IOMUXC_GPIO_AD_08_LPUART2_CTS_B
IOMUXC_GPIO_AD_08_GPT2_COMPARE3
IOMUXC_GPIO_AD_08_GPIOMUX_IO22
IOMUXC_GPIO_AD_08_EWM_OUT_B
IOMUXC_GPIO_AD_08_JTAG_TRSTB
IOMUXC_GPIO_AD_07_LPI2C2_SDA
IOMUXC_GPIO_AD_07_LPUART3_RXD
IOMUXC_GPIO_AD_07_ARM_CM7_RXEV
IOMUXC_GPIO_AD_07_LPUART2_RTS_B
IOMUXC_GPIO_AD_07_GPT2_CAPTURE2
IOMUXC_GPIO_AD_07_GPIOMUX_IO21
IOMUXC_GPIO_AD_07_OCOTP_FUSE_LATCHED
IOMUXC_GPIO_AD_07_XBAR1_INOUT03
IOMUXC_GPIO_AD_06_LPSPI1_SCK
IOMUXC_GPIO_AD_06_PIT_TRIGGER00
IOMUXC_GPIO_AD_06_FLEXPWM1_PWM3_A
IOMUXC_GPIO_AD_06_KPP_COL01
IOMUXC_GPIO_AD_06_GPT2_COMPARE2
IOMUXC_GPIO_AD_06_GPIOMUX_IO20
IOMUXC_GPIO_AD_06_LPI2C1_HREQ
IOMUXC_GPIO_AD_05_LPSPI1_PCS0
IOMUXC_GPIO_AD_05_PIT_TRIGGER01
IOMUXC_GPIO_AD_05_FLEXPWM1_PWM3_B
IOMUXC_GPIO_AD_05_KPP_ROW01
IOMUXC_GPIO_AD_05_GPT2_CAPTURE1
IOMUXC_GPIO_AD_05_GPIOMUX_IO19
IOMUXC_GPIO_AD_04_LPSPI1_SDO
IOMUXC_GPIO_AD_04_PIT_TRIGGER02
IOMUXC_GPIO_AD_04_FLEXPWM1_PWM2_A
IOMUXC_GPIO_AD_04_KPP_COL02
IOMUXC_GPIO_AD_04_GPT2_COMPARE1
IOMUXC_GPIO_AD_04_GPIOMUX_IO18
IOMUXC_GPIO_AD_04_SNVS_VIO_5_CTL

IOMUXC_GPIO_AD_03_LPSPI1_SDI
IOMUXC_GPIO_AD_03_PIT_TRIGGER03
IOMUXC_GPIO_AD_03_FLEXPWM1_PWM2_B
IOMUXC_GPIO_AD_03_KPP_ROW02
IOMUXC_GPIO_AD_03_GPT2_CLK
IOMUXC_GPIO_AD_03_GPIOMUX_IO17
IOMUXC_GPIO_AD_03_SNVS_VIO_5_B
IOMUXC_GPIO_AD_03_JTAG_DE_B
IOMUXC_GPIO_AD_02_LPUART4_TXD
IOMUXC_GPIO_AD_02_LPSPI1_PCS1
IOMUXC_GPIO_AD_02_WDOG2_B
IOMUXC_GPIO_AD_02_LPI2C2_SCL
IOMUXC_GPIO_AD_02_MQS_RIGHT
IOMUXC_GPIO_AD_02_GPIOMUX_IO16
IOMUXC_GPIO_AD_02_ARM_TRACE_CLK
IOMUXC_GPIO_AD_01_LPUART4_RXD
IOMUXC_GPIO_AD_01_LPSPI2_PCS1
IOMUXC_GPIO_AD_01_WDOG1_ANY
IOMUXC_GPIO_AD_01_LPI2C2_SDA
IOMUXC_GPIO_AD_01_MQS_LEFT
IOMUXC_GPIO_AD_01_GPIOMUX_IO15
IOMUXC_GPIO_AD_01_USB_OTG1_OC
IOMUXC_GPIO_AD_01_ARM_TRACE_SWO
IOMUXC_GPIO_AD_00_LPUART2_TXD
IOMUXC_GPIO_AD_00_LPSPI1_PCS2
IOMUXC_GPIO_AD_00_KPP_COL03
IOMUXC_GPIO_AD_00_USB_OTG1_PWR
IOMUXC_GPIO_AD_00_FLEXIO1_IO20
IOMUXC_GPIO_AD_00_GPIOMUX_IO14
IOMUXC_GPIO_AD_00_ARM_NMI
IOMUXC_GPIO_AD_00_ARM_TRACE0
IOMUXC_GPIO_SD_14_FLEXSPI_A_DQS
IOMUXC_GPIO_SD_14_FLEXSPI_B_DQS

IOMUXC_GPIO_SD_13_FLEXSPI_B_SCLK
IOMUXC_GPIO_SD_13_SAI3_RX_BCLK
IOMUXC_GPIO_SD_13_ARM_CM7_TXEV
IOMUXC_GPIO_SD_13_CCM_PMIC_RDY
IOMUXC_GPIO_SD_13_FLEXIO1_IO19
IOMUXC_GPIO_SD_13_GPIO2_IO13
IOMUXC_GPIO_SD_13_SRC_BT_CFG03
IOMUXC_GPIO_SD_12_FLEXSPI_A_DQS
IOMUXC_GPIO_SD_12_LPSPI2_PCS0
IOMUXC_GPIO_SD_12_LPUART1_TXD
IOMUXC_GPIO_SD_12_FLEXIO1_IO18
IOMUXC_GPIO_SD_12_GPIO2_IO12
IOMUXC_GPIO_SD_12_WDOG2_RST_B_DEB
IOMUXC_GPIO_SD_11_FLEXSPI_A_DATA3
IOMUXC_GPIO_SD_11_LPSPI2_SCK
IOMUXC_GPIO_SD_11_LPUART1_RXD
IOMUXC_GPIO_SD_11_FLEXIO1_IO17
IOMUXC_GPIO_SD_11_GPIO2_IO11
IOMUXC_GPIO_SD_11_WDOG1_RST_B_DEB
IOMUXC_GPIO_SD_10_FLEXSPI_A_SCLK
IOMUXC_GPIO_SD_10_LPSPI2_SDO
IOMUXC_GPIO_SD_10_LPUART2_TXD
IOMUXC_GPIO_SD_10_FLEXIO1_IO16
IOMUXC_GPIO_SD_10_GPIO2_IO10
IOMUXC_GPIO_SD_09_FLEXSPI_A_DATA0
IOMUXC_GPIO_SD_09_LPSPI2_SDI
IOMUXC_GPIO_SD_09_LPUART2_RXD
IOMUXC_GPIO_SD_09_FLEXIO1_IO15
IOMUXC_GPIO_SD_09_GPIO2_IO09
IOMUXC_GPIO_SD_08_FLEXSPI_A_DATA2
IOMUXC_GPIO_SD_08_LPI2C2_SCL
IOMUXC_GPIO_SD_08_LPSPI1_SCK
IOMUXC_GPIO_SD_08_FLEXIO1_IO14

IOMUXC_GPIO_SD_08_GPIO2_IO08
IOMUXC_GPIO_SD_07_FLEXSPI_A_DATA1
IOMUXC_GPIO_SD_07_LPI2C2_SDA
IOMUXC_GPIO_SD_07_LPSP1_PCS0
IOMUXC_GPIO_SD_07_FLEXIO1_IO13
IOMUXC_GPIO_SD_07_GPIO2_IO07
IOMUXC_GPIO_SD_06_FLEXSPI_A_SS0_B
IOMUXC_GPIO_SD_06_LPI2C1_SCL
IOMUXC_GPIO_SD_06_LPSP1_SDO
IOMUXC_GPIO_SD_06_FLEXIO1_IO12
IOMUXC_GPIO_SD_06_GPIO2_IO06
IOMUXC_GPIO_SD_05_FLEXSPI_A_SS1_B
IOMUXC_GPIO_SD_05_LPI2C1_SDA
IOMUXC_GPIO_SD_05_LPSP1_SDI
IOMUXC_GPIO_SD_05_FLEXIO1_IO11
IOMUXC_GPIO_SD_05_GPIO2_IO05
IOMUXC_GPIO_SD_04_FLEXPWM1_PWM1_A
IOMUXC_GPIO_SD_04_CCM_WAIT
IOMUXC_GPIO_SD_04_FLEXIO1_IO10
IOMUXC_GPIO_SD_04_GPIO2_IO04
IOMUXC_GPIO_SD_04_SRC_BOOT_MODE00
IOMUXC_GPIO_SD_04_FLEXSPI_B_DATA03
IOMUXC_GPIO_SD_04_SAI3_RX_SYNC
IOMUXC_GPIO_SD_03_FLEXSPI_B_DATA00
IOMUXC_GPIO_SD_03_SAI3_RX_DATA
IOMUXC_GPIO_SD_03_FLEXPWM1_PWM1_B
IOMUXC_GPIO_SD_03_CCM_REF_EN_B
IOMUXC_GPIO_SD_03_FLEXIO1_IO09
IOMUXC_GPIO_SD_03_GPIO2_IO03
IOMUXC_GPIO_SD_03_SRC_BOOT_MODE01
IOMUXC_GPIO_SD_02_FLEXSPI_B_DATA02
IOMUXC_GPIO_SD_02_SAI3_TX_DATA
IOMUXC_GPIO_SD_02_FLEXPWM1_PWM0_A

IOMUXC_GPIO_SD_02_CCM_CLKO1
IOMUXC_GPIO_SD_02_FLEXIO1_IO08
IOMUXC_GPIO_SD_02_GPIO2_IO02
IOMUXC_GPIO_SD_02_SRC_BT_CFG00
IOMUXC_GPIO_SD_01_FLEXSPI_B_DATA01
IOMUXC_GPIO_SD_01_SAI3_TX_BCLK
IOMUXC_GPIO_SD_01_FLEXPWM1_PWM0_B
IOMUXC_GPIO_SD_01_CCM_CLKO2
IOMUXC_GPIO_SD_01_FLEXIO1_IO07
IOMUXC_GPIO_SD_01_GPIO2_IO01
IOMUXC_GPIO_SD_01_SRC_BT_CFG01
IOMUXC_GPIO_SD_00_FLEXSPI_B_SS0_B
IOMUXC_GPIO_SD_00_SAI3_TX_SYNC
IOMUXC_GPIO_SD_00_ARM_CM7_RXEV
IOMUXC_GPIO_SD_00_CCM_STOP
IOMUXC_GPIO_SD_00_FLEXIO1_IO06
IOMUXC_GPIO_SD_00_GPIO2_IO00
IOMUXC_GPIO_SD_00_SRC_BT_CFG02
IOMUXC_GPIO_13_LPUART2_RXD
IOMUXC_GPIO_13_LPSPI2_PCS2
IOMUXC_GPIO_13_KPP_ROW03
IOMUXC_GPIO_13_USB_OTG1_ID
IOMUXC_GPIO_13_FLEXIO1_IO05
IOMUXC_GPIO_13_GPIOMUX_IO13
IOMUXC_GPIO_13_SPDIF_LOCK
IOMUXC_GPIO_13_ARM_TRACE1
IOMUXC_GPIO_12_LPUART3_TXD
IOMUXC_GPIO_12_LPI2C1_SCL
IOMUXC_GPIO_12_KPP_COL00
IOMUXC_GPIO_12_USB_OTG1_OC
IOMUXC_GPIO_12_FLEXIO1_IO04
IOMUXC_GPIO_12_GPIOMUX_IO12
IOMUXC_GPIO_12_SPDIF_EXT_CLK

IOMUXC_GPIO_12_ARM_TRACE2
IOMUXC_GPIO_11_LPUART3_RXD
IOMUXC_GPIO_11_LPI2C1_SDA
IOMUXC_GPIO_11_KPP_ROW00
IOMUXC_GPIO_11_FLEXSPI_B_SS1_B
IOMUXC_GPIO_11_FLEXIO1_IO03
IOMUXC_GPIO_11_GPIOMUX_IO11
IOMUXC_GPIO_11_SPDIF_OUT
IOMUXC_GPIO_11_ARM_TRACE3
IOMUXC_GPIO_10_LPUART1_TXD
IOMUXC_GPIO_10_LPI2C1_HREQ
IOMUXC_GPIO_10_EWM_OUT_B
IOMUXC_GPIO_10_LPI2C2_SCL
IOMUXC_GPIO_10_FLEXIO1_IO02
IOMUXC_GPIO_10_GPIOMUX_IO10
IOMUXC_GPIO_10_SPDIF_IN
IOMUXC_GPIO_09_LPUART1_RXD
IOMUXC_GPIO_09_WDOG1_B
IOMUXC_GPIO_09_FLEXSPI_A_SS1_B
IOMUXC_GPIO_09_LPI2C2_SDA
IOMUXC_GPIO_09_FLEXIO1_IO01
IOMUXC_GPIO_09_GPIOMUX_IO09
IOMUXC_GPIO_09_SPDIF_SR_CLK
IOMUXC_GPIO_08_SAI1_MCLK
IOMUXC_GPIO_08_GPT1_CLK
IOMUXC_GPIO_08_FLEXPWM1_PWM3_A
IOMUXC_GPIO_08_LPUART3_TXD
IOMUXC_GPIO_08_FLEXIO1_IO00
IOMUXC_GPIO_08_GPIOMUX_IO08
IOMUXC_GPIO_08_LPUART1_CTS_B
IOMUXC_GPIO_07_SAI1_TX_SYNC
IOMUXC_GPIO_07_GPT1_COMPARE1
IOMUXC_GPIO_07_FLEXPWM1_PWM3_B

IOMUXC_GPIO_07_LPUART3_RXD
IOMUXC_GPIO_07_SPDIF_LOCK
IOMUXC_GPIO_07_GPIOMUX_IO07
IOMUXC_GPIO_07_LPUART1_RTS_B
IOMUXC_GPIO_06_SAI1_TX_BCLK
IOMUXC_GPIO_06_GPT1_CAPTURE1
IOMUXC_GPIO_06_FLEXPWM1_PWM2_A
IOMUXC_GPIO_06_LPUART4_TXD
IOMUXC_GPIO_06_SPDIF_EXT_CLK
IOMUXC_GPIO_06_GPIOMUX_IO06
IOMUXC_GPIO_05_SAI1_TX_DATA01
IOMUXC_GPIO_05_GPT1_COMPARE2
IOMUXC_GPIO_05_FLEXPWM1_PWM2_B
IOMUXC_GPIO_05_LPUART4_RXD
IOMUXC_GPIO_05_SPDIF_OUT
IOMUXC_GPIO_05_GPIOMUX_IO05
IOMUXC_GPIO_04_SAI1_TX_DATA00
IOMUXC_GPIO_04_GPT1_CAPTURE2
IOMUXC_GPIO_04_FLEXPWM1_PWM1_A
IOMUXC_GPIO_04_SPDIF_IN
IOMUXC_GPIO_04_GPIOMUX_IO04
IOMUXC_GPIO_03_SAI1_RX_DATA00
IOMUXC_GPIO_03_GPT1_COMPARE3
IOMUXC_GPIO_03_FLEXPWM1_PWM1_B
IOMUXC_GPIO_03_SPDIF_SR_CLK
IOMUXC_GPIO_03_GPIOMUX_IO03
IOMUXC_GPIO_02_SAI1_RX_SYNC
IOMUXC_GPIO_02_WDOG2_B
IOMUXC_GPIO_02_FLEXPWM1_PWM0_A
IOMUXC_GPIO_02_LPI2C1_SCL
IOMUXC_GPIO_02_KPP_COL03
IOMUXC_GPIO_02_GPIOMUX_IO02
IOMUXC_GPIO_01_SAI1_RX_BCLK

```
IOMUXC_GPIO_01_WDOG1_ANY
IOMUXC_GPIO_01_FLEXPWM1_PWM0_B
IOMUXC_GPIO_01_LPI2C1_SDA
IOMUXC_GPIO_01_KPP_ROW03
IOMUXC_GPIO_01_GPIOMUX_IO01
IOMUXC_GPIO_00_FLEXSPI_B_DQS
IOMUXC_GPIO_00_SAI3_MCLK
IOMUXC_GPIO_00_LPSPI2_PCS3
IOMUXC_GPIO_00_LPSPI1_PCS3
IOMUXC_GPIO_00_PIT_TRIGGER00
IOMUXC_GPIO_00_GPIOMUX_IO00
IOMUXC_SNVS_PMIC_ON_REQ_SNVS_LP_PMIC_ON_REQ
IOMUXC_SNVS_PMIC_ON_REQ_GPIO5_IO00
IOMUXC_SNVS_TEST_MODE
IOMUXC_SNVS_POR_B
IOMUXC_SNVS_ONOFF
IOMUXC_GPR_SAIMCLK_LOWBITMASK
IOMUXC_GPR_SAIMCLK_HIGHBITMASK
```

```
static inline void IOMUXC_SetPinMux(uint32_t muxRegister, uint32_t muxMode, uint32_t
                                inputRegister, uint32_t inputDaisy, uint32_t
                                configRegister, uint32_t inputOnfield)
```

Sets the IOMUXC pin mux mode.

This is an example to set the PTA6 as the lpuart0_tx:

```
IOMUXC_SetPinMux(IOMUXC_PTA6_LPUART0_TX, 0);
```

This is an example to set the PTA0 as GPIOA0:

```
IOMUXC_SetPinMux(IOMUXC_PTA0_GPIOA0, 0);
```

Note: The first five parameters can be filled with the pin function ID macros.

Parameters

- muxRegister – The pin mux register.
- muxMode – The pin mux mode.
- inputRegister – The select input register.
- inputDaisy – The input daisy.
- configRegister – The config register.
- inputOnfield – Software input on field.

```
static inline void IOMUXC_SetPinConfig(uint32_t muxRegister, uint32_t muxMode, uint32_t
                                     inputRegister, uint32_t inputDaisy, uint32_t
                                     configRegister, uint32_t configValue)
```

Sets the IOMUXC pin configuration.

This is an example to set pin configuration for IOMUXC_PTA3_LPI2C0_SCLS:

```
IOMUXC_SetPinConfig(IOMUXC_PTA3_LPI2C0_SCLS,IOMUXC_SW_PAD_CTL_PAD_PUS_
↵MASK|IOMUXC_SW_PAD_CTL_PAD_PUS(2U))
```

Note: The previous five parameters can be filled with the pin function ID macros.

Parameters

- muxRegister – The pin mux register.
- muxMode – The pin mux mode.
- inputRegister – The select input register.
- inputDaisy – The input daisy.
- configRegister – The config register.
- configValue – The pin config value.

```
static inline void IOMUXC_EnableMode(IOMUXC_GPR_Type *base, uint32_t mode, bool enable)
```

Sets IOMUXC general configuration for some mode.

Parameters

- base – The IOMUXC GPR base address.
- mode – The mode for setting. the mode is the logical OR of “iomuxc_gpr_mode”
- enable – True enable false disable.

```
static inline void IOMUXC_SetSaiMClkClockSource(IOMUXC_GPR_Type *base,
                                                iomuxc_gpr_saimclk_t mclk, uint8_t clkSrc)
```

Sets IOMUXC general configuration for SAI MCLK selection.

Parameters

- base – The IOMUXC GPR base address.
- mclk – The SAI MCLK.
- clkSrc – The clock source. Take refer to register setting details for the clock source in RM.

```
static inline void IOMUXC_MQSEnterSoftwareReset(IOMUXC_GPR_Type *base, bool enable)
```

Enters or exit MQS software reset.

Parameters

- base – The IOMUXC GPR base address.
- enable – Enter or exit MQS software reset.

```
static inline void IOMUXC_MQSEnable(IOMUXC_GPR_Type *base, bool enable)
```

Enables or disables MQS.

Parameters

- base – The IOMUXC GPR base address.

- `enable` – Enable or disable the MQS.

```
static inline void IOMUXC_MQSConfig(IOMUXC_GPR_Type *base,  
                                   iomuxc_mqs_pwm_oversample_rate_t rate, uint8_t  
                                   divider)
```

Configure MQS PWM oversampling rate compared with `mclk` and divider ratio control for `mclk` from `hmclk`.

Parameters

- `base` – The IOMUXC GPR base address.
- `rate` – The MQS PWM oversampling rate, refer to “`iomuxc_mqs_pwm_oversample_rate_t`”.
- `divider` – The divider ratio control for `mclk` from `hmclk`. `mclk freq = 1 / (divider + 1) * hmclk freq`.

FSL_IOMUXC_DRIVER_VERSION

IOMUXC driver version 2.0.3.

FSL_COMPONENT_ID

2.31 KPP: KeyPad Port Driver

```
void KPP_Init(KPP_Type *base, kpp_config_t *configure)
```

KPP initialize. This function ungates the KPP clock and initializes KPP. This function must be called before calling any other KPP driver functions.

Parameters

- `base` – KPP peripheral base address.
- `configure` – The KPP configuration structure pointer.

```
void KPP_Deinit(KPP_Type *base)
```

Deinitializes the KPP module and gates the clock. This function gates the KPP clock. As a result, the KPP module doesn't work after calling this function.

Parameters

- `base` – KPP peripheral base address.

```
static inline void KPP_EnableInterrupts(KPP_Type *base, uint16_t mask)
```

Enable the interrupt.

Parameters

- `base` – KPP peripheral base address.
- `mask` – KPP interrupts to enable. This is a logical OR of the enumeration :: `kpp_interrupt_enable_t`.

```
static inline void KPP_DisableInterrupts(KPP_Type *base, uint16_t mask)
```

Disable the interrupt.

Parameters

- `base` – KPP peripheral base address.
- `mask` – KPP interrupts to disable. This is a logical OR of the enumeration :: `kpp_interrupt_enable_t`.

```
static inline uint16_t KPP_GetStatusFlag(KPP_Type *base)
```

Gets the KPP interrupt event status.

Parameters

- base – KPP peripheral base address.

Returns

The status of the KPP. Application can use the enum type in the “kpp_interrupt_enable_t” to get the right status of the related event.

```
static inline void KPP_ClearStatusFlag(KPP_Type *base, uint16_t mask)
```

Clears KPP status flag.

Parameters

- base – KPP peripheral base address.
- mask – KPP mask to be cleared. This is a logical OR of the enumeration :: kpp_interrupt_enable_t.

```
static inline void KPP_SetSynchronizeChain(KPP_Type *base, uint16_t mask)
```

Set KPP synchronization chain.

Parameters

- base – KPP peripheral base address.
- mask – KPP mask to be cleared. This is a logical OR of the enumeration :: kpp_sync_operation_t.

```
status_t KPP_keyPressScanning(KPP_Type *base, uint8_t *data, uint32_t clockSrc_Hz)
```

Keypad press scanning.

This function will scanning all columns and rows. so all scanning data will be stored in the data pointer.

Parameters

- base – KPP peripheral base address.
- data – KPP key press scanning data. The data buffer should be prepared with length at least equal to KPP_KEYPAD_COLUMNNUM_MAX * KPP_KEYPAD_ROWNUM_MAX. the data pointer is recommended to be a array like uint8_t data[KPP_KEYPAD_COLUMNNUM_MAX]. for example the data[2] = 4, that means in column 1 row 2 has a key press event.
- clockSrc_Hz – Source clock.

Return values

kStatus_Success – kpp press scan succeed.

```
FSL_KPP_DRIVER_VERSION
```

KPP driver version.

```
enum _kpp_interrupt_enable
```

List of interrupts supported by the peripheral. This enumeration uses one-hot encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Values:

enumerator kKPP_keyDepressInterrupt

Keypad depress interrupt source

enumerator kKPP_keyReleaseInterrupt

Keypad release interrupt source

enum `_kpp_sync_operation`

Lists of KPP synchronize chain operation.

Values:

enumerator `kKPP_ClearKeyDepressSyncChain`

Keypad depress interrupt status.

enumerator `kKPP_SetKeyReleasesSyncChain`

Keypad release interrupt status.

typedef enum `_kpp_interrupt_enable` `kpp_interrupt_enable_t`

List of interrupts supported by the peripheral. This enumeration uses one-hot encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

typedef enum `_kpp_sync_operation` `kpp_sync_operation_t`

Lists of KPP synchronize chain operation.

typedef struct `_kpp_config` `kpp_config_t`

Lists of KPP status.

`KPP_KEYPAD_COLUMNNUM_MAX`

`KPP_KEYPAD_ROWNUM_MAX`

struct `_kpp_config`

#include <fsl_kpp.h> Lists of KPP status.

Public Members

uint8_t `activeRow`

The row number: bit 7 ~ 0 represents the row 7 ~ 0.

uint8_t `activeColumn`

The column number: bit 7 ~ 0 represents the column 7 ~ 0.

uint16_t `interrupt`

KPP interrupt source. A logical OR of “`kpp_interrupt_enable_t`”.

2.32 Common Driver

`FSL_COMMON_DRIVER_VERSION`

common driver version.

`DEBUG_CONSOLE_DEVICE_TYPE_NONE`

No debug console.

`DEBUG_CONSOLE_DEVICE_TYPE_UART`

Debug console based on UART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPUART`

Debug console based on LPUART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPSCI`

Debug console based on LPSCI.

`DEBUG_CONSOLE_DEVICE_TYPE_USBCDC`

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_UART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(*addr*, *val*)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(*addr*, *val*)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(*addr*, *bits*)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(*addr*, *bits*)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(*addr*, *bits*)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(*addr*, *clearBits*, *setBits*)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(*addr*, *expected*, *newValue*)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true** , else return **false** .

SDK_ATOMIC_LOCAL_TEST_AND_SET(*addr*, *newValue*)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(*us*, *clockFreqInHz*)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(*count*, *clockFreqInHz*)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

SDK_L1DCACHE_ALIGN(var)

Macro to define a variable with L1 d-cache line size alignment

SDK_SIZEALIGN(var, alignbytes)

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*.

MCUX_CS

AT_CACHE_LINE_SECTION(var)

Define a variable *var*, which is cache line size aligned and be placed in CacheLineData section.

AT_CACHE_LINE_SECTION_INIT(var)

Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

CACHE_LINE_DATA

AT_QUICKACCESS_SECTION_CODE(func)

Place function in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA(var)

Place data in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA_ALIGN(var, alignbytes)

Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

RAMFUNCTION_SECTION_CODE(func)

Place function in ram.

enum __status_groups

Status group numbers.

Values:

enumerator kStatusGroup_Generic
Group number for generic status codes.

enumerator kStatusGroup_FLASH
Group number for FLASH status codes.

enumerator kStatusGroup_LPSPI
Group number for LPSPI status codes.

enumerator kStatusGroup_FLEXIO_SPI
Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI
Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
Group number for LPI2C status codes.

enumerator kStatusGroup_UART
Group number for UART status codes.

enumerator kStatusGroup_I2C
Group number for 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_CODEC
Group number for codec status codes.

enumerator kStatusGroup_ASRC
Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
Group number for codec status codes.

enumerator kStatusGroup_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_QUEUEDSPI
Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
Group number for A-format status codes.

Generic status return codes.

Values:

enumerator kStatus_Success
Generic status for Success.

enumerator kStatus_Fail
Generic status for Fail.

enumerator kStatus_ReadOnly
Generic status for read only failure.

enumerator kStatus_OutOfRange
Generic status for out of range access.

enumerator kStatus_InvalidArgument
Generic status for invalid argument check.

enumerator kStatus_Timeout
Generic status for timeout.

enumerator kStatus_NoTransferInProgress
Generic status for no transfer in progress.

enumerator kStatus_Busy
Generic status for module is busy.

enumerator kStatus_NoData
Generic status for no data is found for the operation.

typedef int32_t status_t
Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)
Allocate memory with given alignment and aligned size.
This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

void SDK_Free(void *ptr)

Free memory.

Parameters

- ptr – The memory to be release.

void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- delayTime_us – Delay time in unit of microsecond.
- coreClock_Hz – Core clock frequency with Hz.

static inline *status_t* EnableIRQ(IRQn_Type interrupt)

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt enabled successfully
- kStatus_Fail – Failed to enable the interrupt

static inline *status_t* DisableIRQ(IRQn_Type interrupt)

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt disabled successfully
- kStatus_Fail – Failed to disable the interrupt

static inline *status_t* EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

`static inline uint32_t DisableGlobalIRQ(void)`

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

`static inline void EnableGlobalIRQ(uint32_t primask)`

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- `primask` – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

`static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t newValue)`

`static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)`

`FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ`

Macro to use the default weak IRQ handler in drivers.

`MAKE_STATUS(group, code)`

Construct a status code value from a group and code number.

`MAKE_VERSION(major, minor, bugfix)`

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused		Major Version		Minor Version		Bug Fix	
31	25	24	17	16	9	8	0

`ARRAY_SIZE(x)`

Computes the number of elements in an array.

`UINT64_H(X)`

Macro to get upper 32 bits of a 64-bit value

`UINT64_L(X)`

Macro to get lower 32 bits of a 64-bit value

`SUPPRESS_FALL_THROUGH_WARNING()`

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING();” need to be added at the end of each case section which misses “break;”statement.

`MSDK_REG_SECURE_ADDR(x)`

Convert the register address to the one used in secure mode.

`MSDK_REG_NONSECURE_ADDR(x)`

Convert the register address to the one used in non-secure mode.

MSDK_HAS_DWT_CYCCNT

The chip supports DWT CYCCNT or not.

MSDK_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.33 Lin_lpuart_driver

FSL_LIN_LPUART_DRIVER_VERSION

LIN LPUART driver version.

enum _lin_lpuart_stop_bit_count

Values:

enumerator kLPUART_OneStopBit

One stop bit

enumerator kLPUART_TwoStopBit

Two stop bits

enum _lin_lpuart_flags

Values:

enumerator kLPUART_TxDataRegEmptyFlag

Transmit data register empty flag, sets when transmit buffer is empty

enumerator kLPUART_TransmissionCompleteFlag

Transmission complete flag, sets when transmission activity complete

enumerator kLPUART_RxDataRegFullFlag

Receive data register full flag, sets when the receive data buffer is full

enumerator kLPUART_IdleLineFlag

Idle line detect flag, sets when idle line detected

enumerator kLPUART_RxOverrunFlag

Receive Overrun, sets when new data is received before data is read from receive register

enumerator kLPUART_NoiseErrorFlag

Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets

enumerator kLPUART_FramingErrorFlag

Frame error flag, sets if logic 0 was detected where stop bit expected

enumerator kLPUART_ParityErrorFlag

If parity enabled, sets upon parity error detection

enumerator kLPUART_LinBreakFlag

LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

enumerator kLPUART_RxActiveEdgeFlag

Receive pin active edge interrupt flag, sets when active edge detected

enumerator kLPUART_RxActiveFlag

Receiver Active Flag (RAF), sets at beginning of valid start bit

enumerator kLPUART_DataMatch1Flag

The next character to be read from LPUART_DATA matches MA1

enumerator kLPUART_DataMatch2Flag

The next character to be read from LPUART_DATA matches MA2

enumerator kLPUART_NoiseErrorInRxDataRegFlag

NOISY bit, sets if noise detected in current data word

enumerator kLPUART_ParityErrorInRxDataRegFlag

PARITY bit, sets if noise detected in current data word

enumerator kLPUART_TxFifoEmptyFlag

TXEMPT bit, sets if transmit buffer is empty

enumerator kLPUART_RxFifoEmptyFlag

RXEMPT bit, sets if receive buffer is empty

enumerator kLPUART_TxFifoOverflowFlag

TXOF bit, sets if transmit buffer overflow occurred

enumerator kLPUART_RxFifoUnderflowFlag

RXUF bit, sets if receive buffer underflow occurred

enum _lin_lpuart_interrupt_enable

Values:

enumerator kLPUART_LinBreakInterruptEnable

LIN break detect.

enumerator kLPUART_RxActiveEdgeInterruptEnable

Receive Active Edge.

enumerator kLPUART_TxDataRegEmptyInterruptEnable

Transmit data register empty.

enumerator kLPUART_TransmissionCompleteInterruptEnable

Transmission complete.

enumerator kLPUART_RxDataRegFullInterruptEnable

Receiver data register full.

enumerator kLPUART_IdleLineInterruptEnable

Idle line.

enumerator kLPUART_RxOverrunInterruptEnable

Receiver Overrun.

enumerator kLPUART_NoiseErrorInterruptEnable

Noise error flag.

enumerator kLPUART_FramingErrorInterruptEnable

Framing error flag.

enumerator kLPUART_ParityErrorInterruptEnable

Parity error flag.

enumerator kLPUART_TxFifoOverflowInterruptEnable

Transmit FIFO Overflow.

enumerator kLPUART_RxFifoUnderflowInterruptEnable

Receive FIFO Underflow.

enum _lin_lpuart_status

Values:

enumerator kStatus_LPUART_TxBusy

TX busy

enumerator kStatus_LPUART_RxBusy

RX busy

enumerator kStatus_LPUART_TxIdle

LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle

LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge

TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge

RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually

Some flag can't manually clear

enumerator kStatus_LPUART_Error

Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun

LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun

LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError

LPUART noise error.

enumerator kStatus_LPUART_FramingError

LPUART framing error.

enumerator kStatus_LPUART_ParityError

LPUART parity error.

enum lin_lpuart_bit_count_per_char_t

Values:

enumerator LPUART_8_BITS_PER_CHAR

8-bit data characters

enumerator LPUART_9_BITS_PER_CHAR

9-bit data characters

enumerator LPUART_10_BITS_PER_CHAR

10-bit data characters

typedef enum _lin_lpuart_stop_bit_count lin_lpuart_stop_bit_count_t

static inline bool LIN_LPUART_GetRxDataPolarity(const LPUART_Type *base)

static inline void LIN_LPUART_SetRxDataPolarity(LPUART_Type *base, bool polarity)

static inline void LIN_LPUART_WriteByte(LPUART_Type *base, uint8_t data)

static inline void LIN_LPUART_ReadByte(const LPUART_Type *base, uint8_t *readData)

status_t LIN_LPUART_CalculateBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)

Calculates the best osr and sbr value for configured baudrate.

Parameters

- base – LPUART peripheral base address
- baudRate_Bps – user configuration structure of type #lin_user_config_t
- srcClock_Hz – pointer to the LIN_LPUART driver state structure
- osr – pointer to osr value
- sbr – pointer to sbr value

Returns

An error code or lin_status_t

void LIN_LPUART_SetBaudRate(LPUART_Type *base, uint32_t *osr, uint16_t *sbr)

Configure baudrate according to osr and sbr value.

Parameters

- base – LPUART peripheral base address
- osr – pointer to osr value
- sbr – pointer to sbr value

lin_status_t LIN_LPUART_Init(LPUART_Type *base, lin_user_config_t *linUserConfig, lin_state_t *linCurrentState, uint32_t linSourceClockFreq)

Initializes an LIN_LPUART instance for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN_LPUART clock source in the application to initialize the LIN_LPUART. This function initializes a LPUART instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, set break field length to be 13 bit times minimum, enable the break detect interrupt, Rx complete interrupt, frame error detect interrupt, and enable the LPUART module transmitter and receiver

Parameters

- base – LPUART peripheral base address
- linUserConfig – user configuration structure of type #lin_user_config_t
- linCurrentState – pointer to the LIN_LPUART driver state structure

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_Deinit(LPUART_Type *base)

Shuts down the LIN_LPUART by disabling interrupts and transmitter/receiver.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendFrameDataBlocking(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize, uint32_t timeoutMSec)

Sends Frame data out through the LIN_LPUART module using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendFrameData(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN_LPUART module using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_GetTransmitStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. This function return LIN_TX_BUSY while sending, or LIN_TIMEOUT if timeout has occurred, or return LIN_SUCCESS when the transmission is complete. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to transmit

Returns

lin_status_t LIN_TX_BUSY, LIN_SUCCESS or LIN_TIMEOUT

lin_status_t LIN_LPUART_RecvFrmDataBlocking(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN_LPUART module using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

`lin_status_t LIN_LPUART_RecvFrmData(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize)`

Receives frame data through the LIN_LPUART module using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Parameters

- `base` – LPUART peripheral base address
- `rxBuff` – buffer containing 8-bit received data
- `rxSize` – the number of bytes to receive

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_AbortTransferData(LPUART_Type *base)`

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GetReceiveStatus(LPUART_Type *base, uint8_t *bytesRemaining)`

Get status of an on-going non-blocking reception. While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function returns the current event ID, `LIN_RX_BUSY` while receiving and return `LIN_SUCCESS`, or timeout (`LIN_TIMEOUT`) when the reception is complete. The `bytesRemaining` shows number of bytes that still needed to receive.

Parameters

- `base` – LPUART peripheral base address
- `bytesRemaining` – Number of bytes still needed to receive

Returns

`lin_status_t` `LIN_RX_BUSY`, `LIN_TIMEOUT` or `LIN_SUCCESS`

`lin_status_t LIN_LPUART_GoToSleepMode(LPUART_Type *base)`

This function puts current node to sleep mode. This function changes current node state to `LIN_NODE_STATE_SLEEP_MODE`.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GotoIdleState(LPUART_Type *base)`

Puts current LIN node to Idle state. This function changes current node state to `LIN_NODE_STATE_IDLE`.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_SendWakeupSignal(LPUART_Type *base)

Sends a wakeup signal through the LIN_LPUART interface.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_MasterSendHeader(LPUART_Type *base, uint8_t id)

Sends frame header out through the LIN_LPUART module using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- base – LPUART peripheral base address
- id – Frame Identifier

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_EnableIRQ(LPUART_Type *base)

Enables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_DisableIRQ(LPUART_Type *base)

Disables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns

lin_status_t

void LIN_LPUART_IRQHandler(LPUART_Type *base)

LIN_LPUART RX TX interrupt handler.

Parameters

- base – LPUART peripheral base address

Returns

void

LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT

Max loops to wait for LPUART transmission complete.

When de-initializing the LIN LPUART module, the program shall wait for the previous transmission to complete. This parameter defines how many loops to check completion before return error. If defined as 0, driver will wait forever until completion.

AUTOBAUD_BAUDRATE_TOLERANCE

BIT_RATE_TOLERANCE_UNSYNC

BIT_DURATION_MAX_19200

BIT_DURATION_MIN_19200

BIT_DURATION_MAX_14400

BIT_DURATION_MIN_14400

BIT_DURATION_MAX_9600

BIT_DURATION_MIN_9600

BIT_DURATION_MAX_4800

BIT_DURATION_MIN_4800

BIT_DURATION_MAX_2400

BIT_DURATION_MIN_2400

TWO_BIT_DURATION_MAX_19200

TWO_BIT_DURATION_MIN_19200

TWO_BIT_DURATION_MAX_14400

TWO_BIT_DURATION_MIN_14400

TWO_BIT_DURATION_MAX_9600

TWO_BIT_DURATION_MIN_9600

TWO_BIT_DURATION_MAX_4800

TWO_BIT_DURATION_MIN_4800

TWO_BIT_DURATION_MAX_2400

TWO_BIT_DURATION_MIN_2400

AUTOBAUD_BREAK_TIME_MIN

2.34 LPI2C: Low Power Inter-Integrated Circuit Driver

void LPI2C_DriverIRQHandler(uint32_t instance)

LPI2C driver IRQ handler common entry.

This function provides the common IRQ request entry for LPI2C.

Parameters

- instance – LPI2C instance.

FSL_LPI2C_DRIVER_VERSION

LPI2C driver version.

LPI2C status return codes.

Values:

enumerator kStatus_LPI2C_Busy

The master is already performing a transfer.

enumerator kStatus_LPI2C_Idle

The slave driver is idle.

enumerator kStatus_LPI2C_Nak

The slave device sent a NAK in response to a byte.

enumerator kStatus_LPI2C_FifoError

FIFO under run or overrun.

enumerator kStatus_LPI2C_BitError

Transferred bit was not seen on the bus.

enumerator kStatus_LPI2C_ArbitrationLost

Arbitration lost error.

enumerator kStatus_LPI2C_PinLowTimeout

SCL or SDA were held low longer than the timeout.

enumerator kStatus_LPI2C_NoTransferInProgress

Attempt to abort a transfer when one is not in progress.

enumerator kStatus_LPI2C_DmaRequestFail

DMA request failed.

enumerator kStatus_LPI2C_Timeout

Timeout polling status flags.

IRQn_Type const kLpi2cIrqs[]

Array to map LPI2C instance number to IRQ number, used internally for LPI2C master interrupt and EDMA transactional APIs.

lpi2c_master_isr_t s_lpi2cMasterIsr

Pointer to master IRQ handler for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

void *s_lpi2cMasterHandle[]

Pointers to master handles for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

uint32_t LPI2C_GetInstance(LPI2C_Type *base)

Returns an instance number given a base address.

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

- base – The LPI2C peripheral base address.

Returns

LPI2C instance number starting from 0.

I2C_RETRY_TIMES

Retry times for waiting flag.

2.35 LPI2C Master Driver

`void LPI2C_MasterGetDefaultConfig(lpi2c_master_config_t *masterConfig)`

Provides a default configuration for the LPI2C master peripheral.

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->debugEnable       = false;
masterConfig->ignoreAck         = false;
masterConfig->pinConfig          = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz       = 100000U;
masterConfig->busIdleTimeout_ns = 0;
masterConfig->pinLowTimeout_ns  = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable = false;
masterConfig->hostRequest.source  = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `LPI2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `lpi2c_master_config_t`.

`void LPI2C_MasterInit(LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t sourceClock_Hz)`

Initializes the LPI2C master peripheral.

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `LPI2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`void LPI2C_MasterDeinit(LPI2C_Type *base)`

Deinitializes the LPI2C master peripheral.

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

`void LPI2C_MasterConfigureDataMatch(LPI2C_Type *base, const lpi2c_data_match_config_t *matchConfig)`

Configures LPI2C master data match feature.

Parameters

- `base` – The LPI2C peripheral base address.
- `matchConfig` – Settings for the data match feature.

status_t LPI2C_MasterCheckAndClearError(LPI2C_Type *base, uint32_t status)

Convert provided flags to status code, and clear any errors if present.

Parameters

- base – The LPI2C peripheral base address.
- status – Current status flags value that will be checked.

Return values

- kStatus_Success –
- kStatus_LPI2C_PinLowTimeout –
- kStatus_LPI2C_ArbitrationLost –
- kStatus_LPI2C_Nak –
- kStatus_LPI2C_FifoError –

status_t LPI2C_CheckForBusyBus(LPI2C_Type *base)

Make sure the bus isn't already busy.

A busy bus is allowed if we are the one driving it.

Parameters

- base – The LPI2C peripheral base address.

Return values

- kStatus_Success –
- kStatus_LPI2C_Busy –

static inline void LPI2C_MasterReset(LPI2C_Type *base)

Performs a software reset.

Restores the LPI2C master peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_MasterEnable(LPI2C_Type *base, bool enable)

Enables or disables the LPI2C module as master.

Parameters

- base – The LPI2C peripheral base address.
- enable – Pass true to enable or false to disable the specified LPI2C as master.

static inline uint32_t LPI2C_MasterGetStatusFlags(LPI2C_Type *base)

Gets the LPI2C master status flags.

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

[_lpi2c_master_flags](#)

Parameters

- base – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void LPI2C_MasterClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)

Clears the LPI2C master status flag state.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

See also:

`_lpi2c_master_flags`.

Parameters

- base – The LPI2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_MasterGetStatusFlags()`.

static inline void LPI2C_MasterEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Enables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline void LPI2C_MasterDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Disables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t LPI2C_MasterGetEnabledInterrupts(LPI2C_Type *base)

Returns the set of currently enabled LPI2C master interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

static inline void LPI2C_MasterEnableDMA(LPI2C_Type *base, bool enableTx, bool enableRx)
Enables or disables LPI2C master DMA requests.

Parameters

- `base` – The LPI2C peripheral base address.
- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.

static inline uint32_t LPI2C_MasterGetTxFifoAddress(LPI2C_Type *base)
Gets LPI2C master transmit data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

static inline uint32_t LPI2C_MasterGetRxFifoAddress(LPI2C_Type *base)
Gets LPI2C master receive data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

static inline void LPI2C_MasterSetWatermarks(LPI2C_Type *base, size_t txWords, size_t rxWords)
Sets the watermarks for LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txWords` – Transmit FIFO watermark value in words. The `kLPI2C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO is equal or less than `txWords`. Writing a value equal or greater than the FIFO size is truncated.
- `rxWords` – Receive FIFO watermark value in words. The `kLPI2C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO is greater than `rxWords`. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPI2C_MasterGetFifoCounts(LPI2C_Type *base, size_t *rxCount, size_t *txCount)
Gets the current number of words in the LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.


```
void LPI2C_MasterSetBaudRate(LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t
                             baudRate_Hz)
```

Sets the I2C bus frequency for master transactions.

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Note: Please note that the second parameter is the clock frequency of LPI2C module, the third parameter means user configured bus baudrate, this implementation is different from other I2C drivers which use baudrate configuration as second parameter and source clock frequency as third parameter.

Parameters

- `base` – The LPI2C peripheral base address.
- `sourceClock_Hz` – LPI2C functional clock frequency in Hertz.
- `baudRate_Hz` – Requested bus frequency in Hertz.

```
static inline bool LPI2C_MasterGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

```
status_t LPI2C_MasterStart(LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
```

Sends a START signal and slave address on the I2C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

```
static inline status_t LPI2C_MasterRepeatedStart(LPI2C_Type *base, uint8_t address,
                                                  lpi2c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `LPI2C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

status_t LPI2C_MasterSend(LPI2C_Type *base, void *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_LPI2C_Nak`.

Parameters

- `base` – The LPI2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or over run.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The LPI2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.

- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterStop(LPI2C_Type *base)`

Sends a STOP signal on the I2C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterTransferBlocking(LPI2C_Type *base, lpi2c_master_transfer_t *transfer)`

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- `base` – The LPI2C peripheral base address.
- `transfer` – Pointer to the transfer structure.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`void LPI2C_MasterTransferCreateHandle(LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)`

Creates a new handle for the LPI2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_MasterTransferNonBlocking(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *lpi2c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_LPI2C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t LPI2C_MasterTransferGetCount(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *size_t* *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void LPI2C_MasterTransferAbort(LPI2C_Type *base, *lpi2c_master_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

void LPI2C_MasterTransferHandleIRQ(LPI2C_Type *base, void *lpi2cMasterHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The LPI2C peripheral base address.
- lpi2cMasterHandle – Pointer to the LPI2C master driver handle.

enum _lpi2c_master_flags

LPI2C master peripheral flags.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kLPI2C_MasterTxReadyFlag

Transmit data flag

enumerator kLPI2C_MasterRxReadyFlag

Receive data flag

enumerator kLPI2C_MasterEndOfPacketFlag

End Packet flag

enumerator kLPI2C_MasterStopDetectFlag

Stop detect flag

enumerator kLPI2C_MasterNackDetectFlag

NACK detect flag

enumerator kLPI2C_MasterArbitrationLostFlag

Arbitration lost flag

enumerator kLPI2C_MasterFifoErrFlag

FIFO error flag

enumerator kLPI2C_MasterPinLowTimeoutFlag

Pin low timeout flag

enumerator kLPI2C_MasterDataMatchFlag

Data match flag

enumerator kLPI2C_MasterBusyFlag

Master busy flag

enumerator kLPI2C_MasterBusBusyFlag

Bus busy flag

enumerator kLPI2C_MasterClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_MasterIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_MasterErrorFlags

Errors to check for.

enum _lpi2c_direction

Direction of master and slave transfers.

Values:

enumerator kLPI2C_Write

Master transmit.

enumerator kLPI2C_Read

Master receive.

enum _lpi2c_master_pin_config

LPI2C pin configuration.

Values:

enumerator kLPI2C_2PinOpenDrain

LPI2C Configured for 2-pin open drain mode

enumerator kLPI2C_2PinOutputOnly

LPI2C Configured for 2-pin output only mode (ultra-fast mode)

enumerator kLPI2C_2PinPushPull

LPI2C Configured for 2-pin push-pull mode

enumerator kLPI2C_4PinPushPull

LPI2C Configured for 4-pin push-pull mode

enumerator kLPI2C_2PinOpenDrainWithSeparateSlave

LPI2C Configured for 2-pin open drain mode with separate LPI2C slave

enumerator kLPI2C_2PinOutputOnlyWithSeparateSlave

LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave

enumerator kLPI2C_2PinPushPullWithSeparateSlave

LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave

enumerator kLPI2C_4PinPushPullWithInvertedOutput

LPI2C Configured for 4-pin push-pull mode(inverted outputs)

enum _lpi2c_host_request_source

LPI2C master host request selection.

Values:

enumerator kLPI2C_HostRequestExternalPin

Select the LPI2C_HREQ pin as the host request input

enumerator kLPI2C_HostRequestInputTrigger

Select the input trigger as the host request input

enum _lpi2c_host_request_polarity

LPI2C master host request pin polarity configuration.

Values:

enumerator kLPI2C_HostRequestPinActiveLow

Configure the LPI2C_HREQ pin active low

enumerator kLPI2C_HostRequestPinActiveHigh

Configure the LPI2C_HREQ pin active high

enum _lpi2c_data_match_config_mode

LPI2C master data match configuration modes.

Values:

enumerator kLPI2C_MatchDisabled

LPI2C Match Disabled

enumerator kLPI2C_1stWordEqualsM0OrM1

LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1

enumerator kLPI2C_AnyWordEqualsM0OrM1

LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1

enumerator kLPI2C_1stWordEqualsM0And2ndWordEqualsM1

LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1

enumerator kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1

LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1

enumerator kLPI2C_1stWordAndM1EqualsM0AndM1

LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1

enumerator kLPI2C_AnyWordAndM1EqualsM0AndM1

LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1

enum _lpi2c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_lpi2c_master_transfer::flags` field.

Values:

enumerator kLPI2C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kLPI2C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kLPI2C_TransferRepeatedStartFlag

Send a repeated start condition

enumerator kLPI2C_TransferNoStopFlag

Don't send a stop condition.

```
typedef enum _lpi2c_direction lpi2c_direction_t
```

Direction of master and slave transfers.

```
typedef enum _lpi2c_master_pin_config lpi2c_master_pin_config_t
```

LPI2C pin configuration.

```
typedef enum _lpi2c_host_request_source lpi2c_host_request_source_t
```

LPI2C master host request selection.

```
typedef enum _lpi2c_host_request_polarity lpi2c_host_request_polarity_t
```

LPI2C master host request pin polarity configuration.

```
typedef struct _lpi2c_master_config lpi2c_master_config_t
```

Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _lpi2c_data_match_config_mode lpi2c_data_match_config_mode_t
```

LPI2C master data match configuration modes.

```
typedef struct _lpi2c_match_config lpi2c_data_match_config_t
```

LPI2C master data match configuration structure.

```
typedef struct _lpi2c_master_transfer lpi2c_master_transfer_t
```

LPI2C master descriptor of the transfer.

```
typedef struct _lpi2c_master_handle lpi2c_master_handle_t
```

LPI2C master handle of the transfer.

```
typedef void (*lpi2c_master_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterTransferCreateHandle().

Param base

The LPI2C peripheral base address.

Param handle

Pointer to the LPI2C master driver handle.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef void (*lpi2c_master_isr_t)(LPI2C_Type *base, void *handle)
```

Typedef for master interrupt handler, used internally for LPI2C master interrupt and EDMA transactional APIs.

```
struct _lpi2c_master_config
```

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

bool enableDoze

Whether master is enabled in doze mode.

bool debugEnable

Enable transfers to continue when halted in debug mode.

bool ignoreAck

Whether to ignore ACK/NACK.

lpi2c_master_pin_config_t pinConfig

The pin configuration option.

uint32_t baudRate_Hz

Desired baud rate in Hertz.

uint32_t busIdleTimeout_ns

Bus idle timeout in nanoseconds. Set to 0 to disable.

uint32_t pinLowTimeout_ns

Pin low timeout in nanoseconds. Set to 0 to disable.

uint8_t sdaGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SDA pin. Set to 0 to disable.

uint8_t sclGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SCL pin. Set to 0 to disable.

struct *_lpi2c_master_config* hostRequest

Host request options.

struct *_lpi2c_match_config*

#include <fsl_lpi2c.h> LPI2C master data match configuration structure.

Public Members

lpi2c_data_match_config_mode_t matchMode

Data match configuration setting.

bool rxDataMatchOnly

When set to true, received data is ignored until a successful match.

uint32_t match0

Match value 0.

uint32_t match1

Match value 1.

struct *_lpi2c_master_transfer*

#include <fsl_lpi2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_lpi2c_master_transfer_flags` for available options. Set to 0 or `kLPI2C_TransferDefaultFlag` for normal transfers.

uint16_t slaveAddress

The 7-bit slave address.

lpi2c_direction_t direction

Either `kLPI2C_Read` or `kLPI2C_Write`.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct `_lpi2c_master_handle`

#include <fsl_lpi2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint16_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

uint16_t commandBuffer[6]

LPI2C command sequence. When all 6 command words are used: `Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word]`

lpi2c_master_transfer_t transfer

Copy of the current transfer info.

lpi2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData

Application data passed to callback.

struct hostRequest

Public Members

`bool enable`

Enable host request.

`lpi2c_host_request_source_t source`

Host request source.

`lpi2c_host_request_polarity_t polarity`

Host request pin polarity.

2.36 LPI2C Master DMA Driver

```
void LPI2C_MasterCreateEDMAHandle(LPI2C_Type *base, lpi2c_master_edma_handle_t *handle,
                                   edma_handle_t *rxDmaHandle, edma_handle_t
                                   *txDmaHandle, lpi2c_master_edma_transfer_callback_t
                                   callback, void *userData)
```

Create a new handle for the LPI2C master DMA APIs.

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_MasterTransferAbortEDMA()` API shall be called.

For devices where the LPI2C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – **[out]** Pointer to the LPI2C master driver handle.
- *rxDmaHandle* – Handle for the eDMA receive channel. Created by the user prior to calling this function.
- *txDmaHandle* – Handle for the eDMA transmit channel. Created by the user prior to calling this function.
- *callback* – User provided pointer to the asynchronous callback function.
- *userData* – User provided pointer to the application callback data.

```
status_t LPI2C_MasterTransferEDMA(LPI2C_Type *base, lpi2c_master_edma_handle_t *handle,
                                   lpi2c_master_transfer_t *transfer)
```

Performs a non-blocking DMA-based transaction on the I2C bus.

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to the LPI2C master driver handle.
- *transfer* – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_LPI2C_Busy` – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

status_t LPI2C_MasterTransferGetCountEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a DMA transaction currently in progress.

status_t LPI2C_MasterTransferAbortEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the eDMA peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

Return values

- kStatus_Success – A transaction was successfully aborted.
- kStatus_LPI2C_Idle – There is not a DMA transaction currently in progress.

typedef struct *lpi2c_master_edma_handle* *lpi2c_master_edma_handle_t*
LPI2C master EDMA handle of the transfer.

typedef void (**lpi2c_master_edma_transfer_callback_t*)(LPI2C_Type *base,
lpi2c_master_edma_handle_t *handle, *status_t* completionStatus, void *userData)

Master DMA completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterCreateEDMAHandle().

Param base

The LPI2C peripheral base address.

Param handle

Handle associated with the completed transfer.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
struct _lpi2c_master_edma_handle
    #include <fsl_lpi2c_edma.h> Driver handle for master DMA APIs.
```

Note: The contents of this structure are private and subject to change.

Public Members

LPI2C_Type *base
LPI2C base pointer.

bool isBusy
Transfer state machine current state.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

uint16_t commandBuffer[20]
LPI2C command sequence. When all 10 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word] + receive&Size[4 words]

lpi2c_master_transfer_t transfer
Copy of the current transfer info.

lpi2c_master_edma_transfer_callback_t completionCallback
Callback function pointer.

void *userData
Application data passed to callback.

edma_handle_t *rx
Handle for receive DMA channel.

edma_handle_t *tx
Handle for transmit DMA channel.

edma_tcd_t tcds[3]
Software TCD. Three are allocated to provide enough room to align to 32-bytes.

2.37 LPI2C Slave Driver

```
void LPI2C_SlaveGetDefaultConfig(lpi2c_slave_config_t *slaveConfig)
```

Provides a default configuration for the LPI2C slave peripheral.

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave      = true;
slaveConfig->address0         = 0U;
slaveConfig->address1         = 0U;
slaveConfig->addressMatchMode = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable = true;
slaveConfig->filterEnable     = true;
slaveConfig->enableGeneralCall = false;
slaveConfig->sclStall.enableAck = false;
slaveConfig->sclStall.enableTx  = true;
slaveConfig->sclStall.enableRx  = true;
```

(continues on next page)

(continued from previous page)

```

slaveConfig->sclStall.enableAddress    = true;
slaveConfig->ignoreAck                 = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns   = 0;
slaveConfig->sclGlitchFilterWidth_ns   = 0;
slaveConfig->dataValidDelay_ns         = 0;
slaveConfig->clockHoldTime_ns          = 0;

```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `LPI2C_SlaveInit()`. Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure that is set to default values. Refer to `lpi2c_slave_config_t`.

```
void LPI2C_SlaveInit(LPI2C_Type *base, const lpi2c_slave_config_t *slaveConfig, uint32_t
                    sourceClock_Hz)
```

Initializes the LPI2C slave peripheral.

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `LPI2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

```
void LPI2C_SlaveDeinit(LPI2C_Type *base)
```

Deinitializes the LPI2C slave peripheral.

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveReset(LPI2C_Type *base)
```

Performs a software reset of the LPI2C slave peripheral.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveEnable(LPI2C_Type *base, bool enable)
```

Enables or disables the LPI2C module as slave.

Parameters

- `base` – The LPI2C peripheral base address.
- `enable` – Pass true to enable or false to disable the specified LPI2C as slave.

```
static inline uint32_t LPI2C_SlaveGetStatusFlags(LPI2C_Type *base)
```

Gets the LPI2C slave status flags.

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_lpi2c_slave_flags`

Parameters

- `base` – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void LPI2C_SlaveClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)
```

Clears the LPI2C status flag state.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

Attempts to clear other flags has no effect.

See also:

`_lpi2c_slave_flags`.

Parameters

- `base` – The LPI2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_SlaveGetStatusFlags()`.

```
static inline void LPI2C_SlaveEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Enables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void LPI2C_SlaveDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Disables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t LPI2C_SlaveGetEnabledInterrupts(LPI2C_Type *base)
```

Returns the set of currently enabled LPI2C slave interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline void LPI2C_SlaveEnableDMA(LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)
```

Enables or disables the LPI2C slave peripheral DMA requests.

Parameters

- base – The LPI2C peripheral base address.
- enableAddressValid – Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
- enableRx – Enable flag for the receive data DMA request. Pass true for enable, false for disable.
- enableTx – Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

```
static inline bool LPI2C_SlaveGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the slave mode to be enabled.

Parameters

- base – The LPI2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

```
static inline void LPI2C_SlaveTransmitAck(LPI2C_Type *base, bool ackOrNack)
```

Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.

Use this function to send an ACK or NAK when the `kLPI2C_SlaveTransmitAckFlag` is asserted. This only happens if you enable the `sclStall.enableAck` field of the `lpi2c_slave_config_t` configuration structure used to initialize the slave peripheral.

Parameters

- base – The LPI2C peripheral base address.
- ackOrNack – Pass true for an ACK or false for a NAK.

```
static inline void LPI2C_SlaveEnableAckStall(LPI2C_Type *base, bool enable)
```

Enables or disables ACKSTALL.

When enables ACKSTALL, software can transmit either an ACK or NAK on the I2C bus in response to a byte from the master.

Parameters

- base – The LPI2C peripheral base address.
- enable – True will enable ACKSTALL, false will disable ACKSTALL.

static inline uint32_t LPI2C_SlaveGetReceivedAddress(LPI2C_Type *base)

Returns the slave address sent by the I2C master.

This function should only be called if the kLPI2C_SlaveAddressValidFlag is asserted.

Parameters

- base – The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

status_t LPI2C_SlaveSend(LPI2C_Type *base, void *txBuff, size_t txSize, size_t *actualTxSize)

Performs a polling send transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- actualTxSize – **[out]**

Returns

Error or success status returned by API.

status_t LPI2C_SlaveReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.
- actualRxSize – **[out]**

Returns

Error or success status returned by API.

void LPI2C_SlaveTransferCreateHandle(LPI2C_Type *base, lpi2c_slave_handle_t *handle, lpi2c_slave_transfer_callback_t callback, void *userData)

Creates a new handle for the LPI2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_SlaveTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

```
status_t LPI2C_SlaveTransferNonBlocking(LPI2C_Type *base, lpi2c_slave_handle_t *handle,
                                         uint32_t eventMask)
```

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of *lpi2c_slave_transfer_event_t* enumerators for the events you wish to receive. The *kLPI2C_SlaveTransmitEvent* and *kLPI2C_SlaveReceiveEvent* events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the *kLPI2C_SlaveAllEvents* constant is provided as a convenient way to enable all events.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *lpi2c_slave_handle_t* structure which stores the transfer state.
- *eventMask* – Bit mask formed by OR'ing together *lpi2c_slave_transfer_event_t* enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and *kLPI2C_SlaveAllEvents* to enable all events.

Return values

- *kStatus__Success* – Slave transfers were successfully started.
- *kStatus__LPI2C_Busy* – Slave transfers have already been started on this handle.

```
status_t LPI2C_SlaveTransferGetCount(LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t
                                      *count)
```

Gets the slave transfer status during a non-blocking transfer.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *i2c_slave_handle_t* structure.
- *count* – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- *kStatus__Success* –
- *kStatus__NoTransferInProgress* –

```
void LPI2C_SlaveTransferAbort(LPI2C_Type *base, lpi2c_slave_handle_t *handle)
```

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- *base* – The LPI2C peripheral base address.

- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`void LPI2C_SlaveTransferHandleIRQ(LPI2C_Type *base, lpi2c_slave_handle_t *handle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The LPI2C peripheral base address.
- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`enum _lpi2c_slave_flags`

LPI2C slave peripheral flags.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator `kLPI2C_SlaveTxReadyFlag`

Transmit data flag

enumerator `kLPI2C_SlaveRxReadyFlag`

Receive data flag

enumerator `kLPI2C_SlaveAddressValidFlag`

Address valid flag

enumerator `kLPI2C_SlaveTransmitAckFlag`

Transmit ACK flag

enumerator `kLPI2C_SlaveRepeatedStartDetectFlag`

Repeated start detect flag

enumerator `kLPI2C_SlaveStopDetectFlag`

Stop detect flag

enumerator `kLPI2C_SlaveBitErrFlag`

Bit error flag

enumerator `kLPI2C_SlaveFifoErrFlag`

FIFO error flag

enumerator `kLPI2C_SlaveAddressMatch0Flag`

Address match 0 flag

enumerator kLPI2C_SlaveAddressMatch1Flag

Address match 1 flag

enumerator kLPI2C_SlaveGeneralCallFlag

General call flag

enumerator kLPI2C_SlaveBusyFlag

Master busy flag

enumerator kLPI2C_SlaveBusBusyFlag

Bus busy flag

enumerator kLPI2C_SlaveClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_SlaveIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_SlaveErrorFlags

Errors to check for.

enum _lpi2c_slave_address_match

LPI2C slave address match options.

Values:

enumerator kLPI2C_MatchAddress0

Match only address 0.

enumerator kLPI2C_MatchAddress0OrAddress1

Match either address 0 or address 1.

enumerator kLPI2C_MatchAddress0ThroughAddress1

Match a range of slave addresses from address 0 through address 1.

enum _lpi2c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kLPI2C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kLPI2C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kLPI2C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kLPI2C_SlaveTransmitAckEvent

Callback needs to either transmit an ACK or NACK.

enumerator kLPI2C_SlaveRepeatedStartEvent

A repeated start was detected.

enumerator kLPI2C_SlaveCompletionEvent

A stop was detected, completing the transfer.

enumerator kLPI2C_SlaveAllEvents

Bit mask of all available events.

typedef enum *_lpi2c_slave_address_match* lpi2c_slave_address_match_t

LPI2C slave address match options.

typedef struct *_lpi2c_slave_config* lpi2c_slave_config_t

Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_lpi2c_slave_transfer_event* lpi2c_slave_transfer_event_t

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct *_lpi2c_slave_transfer* lpi2c_slave_transfer_t

LPI2C slave transfer structure.

typedef struct *_lpi2c_slave_handle* lpi2c_slave_handle_t

LPI2C slave handle structure.

typedef void (*lpi2c_slave_transfer_callback_t)(LPI2C_Type *base, *lpi2c_slave_transfer_t* *transfer, void *userData)

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the LPI2C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

struct *_lpi2c_slave_config*

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

`bool enableSlave`
Enable slave mode.

`uint8_t address0`
Slave's 7-bit address.

`uint8_t address1`
Alternate slave 7-bit address.

`lpi2c_slave_address_match_t addressMatchMode`
Address matching options.

`bool filterDozeEnable`
Enable digital glitch filter in doze mode.

`bool filterEnable`
Enable digital glitch filter.

`bool enableGeneralCall`
Enable general call address matching.

`struct _lpi2c_slave_config sclStall`
SCL stall enable options.

`bool ignoreAck`
Continue transfers after a NACK is detected.

`bool enableReceivedAddressRead`
Enable reading the address received address as the first byte of data.

`uint32_t sdaGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SDA signal. Set to 0 to disable.

`uint32_t sclGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SCL signal. Set to 0 to disable.

`uint32_t dataValidDelay_ns`
Width in nanoseconds of the data valid delay.

`uint32_t clockHoldTime_ns`
Width in nanoseconds of the clock hold time.

`struct _lpi2c_slave_transfer`
#include <fsl_lpi2c.h> LPI2C slave transfer structure.

Public Members

`lpi2c_slave_transfer_event_t event`
Reason the callback is being invoked.

`uint8_t receivedAddress`
Matching address send by master.

`uint8_t *data`
Transfer buffer

`size_t dataSize`
Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kLPI2C_SlaveCompletionEvent.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct *_lpi2c_slave_handle*

#include <fsl_lpi2c.h> LPI2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

lpi2c_slave_transfer_t transfer

LPI2C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

lpi2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

struct sclStall

Public Members

bool enableAck

Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted. Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

bool enableTx

Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.

bool enableRx

Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.

bool enableAddress

Enables SCL clock stretching when the address valid flag is asserted.

2.38 LPSPI: Low Power Serial Peripheral Interface

2.39 LPSPI Peripheral driver

```
void LPSPI_MasterInit(LPSPI_Type *base, const lpspi_master_config_t *masterConfig, uint32_t srcClock_Hz)
```

Initializes the LPSPI master.

Parameters

- base – LPSPI peripheral address.
- masterConfig – Pointer to structure *lpspi_master_config_t*.
- srcClock_Hz – Module source input clock in Hertz

```
void LPSPI_MasterGetDefaultConfig(lpspi_master_config_t *masterConfig)
```

Sets the *lpspi_master_config_t* structure to default values.

This API initializes the configuration structure for LPSPI_MasterInit(). The initialized structure can remain unchanged in LPSPI_MasterInit(), or can be modified before calling the LPSPI_MasterInit(). Example:

```
lpspi_master_config_t masterConfig;  
LPSPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – pointer to *lpspi_master_config_t* structure

```
void LPSPI_SlaveInit(LPSPI_Type *base, const lpspi_slave_config_t *slaveConfig)
```

LPSPI slave configuration.

Parameters

- base – LPSPI peripheral address.
- slaveConfig – Pointer to a structure *lpspi_slave_config_t*.

```
void LPSPI_SlaveGetDefaultConfig(lpspi_slave_config_t *slaveConfig)
```

Sets the *lpspi_slave_config_t* structure to default values.

This API initializes the configuration structure for LPSPI_SlaveInit(). The initialized structure can remain unchanged in LPSPI_SlaveInit() or can be modified before calling the LPSPI_SlaveInit(). Example:

```
lpspi_slave_config_t slaveConfig;  
LPSPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – pointer to *lpspi_slave_config_t* structure.

```
void LPSPI_Deinit(LPSPI_Type *base)
```

De-initializes the LPSPI peripheral. Call this API to disable the LPSPI clock.

Parameters

- base – LPSPI peripheral address.

```
void LPSPI_Reset(LPSPI_Type *base)
```

Restores the LPSPI peripheral to reset state. Note that this function sets all registers to reset state. As a result, the LPSPI module can't work after calling this API.

Parameters

- base – LPSPI peripheral address.

uint32_t LPSPI_GetInstance(LPSPI_Type *base)

Get the LPSPI instance from peripheral base address.

Parameters

- base – LPSPI peripheral base address.

Returns

LPSPI instance.

static inline void LPSPI_Enable(LPSPI_Type *base, bool enable)

Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Parameters

- base – LPSPI peripheral address.
- enable – Pass true to enable module, false to disable module.

static inline uint32_t LPSPI_GetStatusFlags(LPSPI_Type *base)

Gets the LPSPI status flag state.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI status(in SR register).

static inline uint8_t LPSPI_GetTxFifoSize(LPSPI_Type *base)

Gets the LPSPI Tx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Tx FIFO size.

static inline uint8_t LPSPI_GetRxFifoSize(LPSPI_Type *base)

Gets the LPSPI Rx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Rx FIFO size.

static inline uint32_t LPSPI_GetTxFifoCount(LPSPI_Type *base)

Gets the LPSPI Tx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the transmit FIFO.

static inline uint32_t LPSPI_GetRxFifoCount(LPSPI_Type *base)

Gets the LPSPI Rx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the receive FIFO.

static inline void LPSPI_ClearStatusFlags(LPSPI_Type *base, uint32_t statusFlags)

Clears the LPSPI status flag.

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the `_lpspi_flags`. Example usage:

```
LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag | kLPSPI_RxDataReadyFlag);
```

Parameters

- base – LPSPI peripheral address.
- statusFlags – The status flag used from type `_lpspi_flags`.

static inline uint32_t LPSPI_GetTcr(LPSPI_Type *base)

static inline void LPSPI_EnableInterrupts(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI interrupts.

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_DisableInterrupts(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI interrupts.

```
LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_EnableDMA(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

static inline void LPSPI_DisableDMA(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
SPI_DisableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

```
static inline uint32_t LPSPI_GetTxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Transmit Data Register address for a DMA operation.

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Transmit Data Register address.

```
static inline uint32_t LPSPI_GetRxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Receive Data Register address for a DMA operation.

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Receive Data Register address.

```
bool LPSPI_CheckTransferArgument(LPSPI_Type *base, lpspi_transfer_t *transfer, bool isEdma)
```

Check the argument for transfer .

Parameters

- base – LPSPI peripheral address.
- transfer – the transfer struct to be used.
- isEdma – True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns

Return true for right and false for wrong.

```
static inline void LPSPI_SetMasterSlaveMode(LPSPI_Type *base, lpspi_master_slave_mode_t mode)
```

Configures the LPSPI for either master or slave.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

- base – LPSPI peripheral address.
- mode – Mode setting (master or slave) of type `lpspi_master_slave_mode_t`.

```
static inline void LPSPI_SelectTransferPCS(LPSPI_Type *base, lpspi_which_pcs_t select)
```

Configures the peripheral chip select used for the transfer.

Parameters

- base – LPSPI peripheral address.

- select – LPSPI Peripheral Chip Select (PCS) configuration.

static inline void LPSPI_SetPCSContinuous(LPSPI_Type *base, bool IsContinuous)

Set the PCS signal to continuous or uncontinuous mode.

Note: In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

- base – LPSPI peripheral address.
- IsContinuous – True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

static inline bool LPSPI_IsMaster(LPSPI_Type *base)

Returns whether the LPSPI module is in master mode.

Parameters

- base – LPSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

static inline void LPSPI_FlushFifo(LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)

Flushes the LPSPI FIFOs.

Parameters

- base – LPSPI peripheral address.
- flushTxFifo – Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
- flushRxFifo – Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

static inline void LPSPI_SetFifoWatermarks(LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)

Sets the transmit and receive FIFO watermark values.

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

- base – LPSPI peripheral address.
- txWater – The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
- rxWater – The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPSPI_SetAllPcsPolarity(LPSPI_Type *base, uint32_t mask)

Configures all LPSPI peripheral chip select polarities simultaneously.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow | kLPSPI_Pcs1ActiveLow);
```

Parameters

- base – LPSPI peripheral address.
- mask – The PCS polarity mask; Use the enum `_lpspi_pcs_polarity`.

```
static inline void LPSPI_SetFrameSize(LPSPI_Type *base, uint32_t frameSize)
```

Configures the frame size.

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPi in slave mode, although the command register does not update until after the LPSPi is enabled. After it is enabled, the transmit command register should only be changed if the LPSPi is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

- base – LPSPI peripheral address.
- frameSize – The frame size in number of bits.

```
uint32_t LPSPI_MasterSetBaudRate(LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t  
srcClock_Hz, uint32_t *tcrPrescaleValue)
```

Sets the LPSPi baud rate in bits per second.

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale `tcrPrescaleValue` parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPi module must first be disabled before configuring this. Note that the LPSPi module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- baudRate_Bps – The desired baud rate in bits per second.
- srcClock_Hz – Module source input clock in Hertz.
- tcrPrescaleValue – The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a “0” if the LPSPi is not configured for master mode or if the LPSPi module is not disabled.

```
void LPSPI_MasterSetDelayScaler(LPSPI_Type *base, uint32_t scaler, lpspi_delay_type_t
                               whichDelay)
```

Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- `base` – LPSPI peripheral address.
- `scaler` – The 8-bit delay value 0x00 to 0xFF (255).
- `whichDelay` – The desired delay to configure, must be of type `lpspi_delay_type_t`.

```
uint32_t LPSPI_MasterSetDelayTimes(LPSPI_Type *base, uint32_t delayTimeInNanoSec,
                                   lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)
```

Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPI module must be configured for master mode before configuring this. And note that the $\text{delayTime} = \text{LPSPI\_clockSource} / (\text{PRESCALE} * \text{Delay\_scaler})$.

Parameters

- `base` – LPSPI peripheral address.
- `delayTimeInNanoSec` – The desired delay value in nano-seconds.
- `whichDelay` – The desired delay to configuration, which must be of type `lpspi_delay_type_t`.
- `srcClock_Hz` – Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

```
static inline void LPSPI_WriteData(LPSPI_Type *base, uint32_t data)
```

Writes data into the transmit data buffer.

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result

in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

- base – LPSPI peripheral address.
- data – The data word to be sent.

```
static inline uint32_t LPSPI_ReadData(LPSPI_Type *base)
```

Reads data from the data buffer.

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The data read from the data buffer.

```
void LPSPI_SetDummyData(LPSPI_Type *base, uint8_t dummyData)
```

Set up the dummy data.

Parameters

- base – LPSPI peripheral address.
- dummyData – Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

```
void LPSPI_MasterTransferCreateHandle(LPSPI_Type *base, lpspi_master_handle_t *handle,  
                                     lpspi_master_transfer_callback_t callback, void  
                                     *userData)
```

Initializes the LPSPI master handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to `lpspi_master_handle_t`.
- callback – DSPI callback.
- userData – callback function parameter.

```
status_t LPSPI_MasterTransferBlocking(LPSPI_Type *base, lpspi_transfer_t *transfer)
```

LPSPI master transfer data using a polling method.

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of status_t.

status_t LPSPI_MasterTransferNonBlocking(*LPSPI_Type* *base, *lpspi_master_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using an interrupt method.

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_master_handle_t* structure which stores the transfer state.
- transfer – pointer to *lpspi_transfer_t* structure.

Returns

status of status_t.

status_t LPSPI_MasterTransferGetCount(*LPSPI_Type* *base, *lpspi_master_handle_t* *handle, *size_t* *count)

Gets the master transfer remaining bytes.

This function gets the master transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_master_handle_t* structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

void LPSPI_MasterTransferAbort(*LPSPI_Type* *base, *lpspi_master_handle_t* *handle)

LPSPI master abort transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_master_handle_t* structure which stores the transfer state.

void LPSPI_MasterTransferHandleIRQ(*LPSPI_Type* *base, *lpspi_master_handle_t* *handle)

LPSPI Master IRQ handler function.

This function processes the LPSPI transmit and receive IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_master_handle_t* structure which stores the transfer state.


```
void LPSPI_SlaveTransferCreateHandle(LPSPI_Type *base, lpspi_slave_handle_t *handle,
                                     lpspi_slave_transfer_callback_t callback, void *userData)
```

Initializes the LPSPI slave handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to *lpspi_slave_handle_t*.
- callback – DSPI callback.
- userData – callback function parameter.

```
status_t LPSPI_SlaveTransferNonBlocking(LPSPI_Type *base, lpspi_slave_handle_t *handle,
                                         lpspi_transfer_t *transfer)
```

LPSPI slave transfer data using an interrupt method.

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_slave_handle_t* structure which stores the transfer state.
- transfer – pointer to *lpspi_transfer_t* structure.

Returns

status of *status_t*.

```
status_t LPSPI_SlaveTransferGetCount(LPSPI_Type *base, lpspi_slave_handle_t *handle, size_t
                                     *count)
```

Gets the slave transfer remaining bytes.

This function gets the slave transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_slave_handle_t* structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of *status_t*.

```
void LPSPI_SlaveTransferAbort(LPSPI_Type *base, lpspi_slave_handle_t *handle)
```

LPSPI slave aborts a transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_slave_handle_t* structure which stores the transfer state.

void LPSPI_SlaveTransferHandleIRQ(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI Slave IRQ handler function.

This function processes the LPSPI transmit and receives an IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to *lpspi_slave_handle_t* structure which stores the transfer state.

bool LPSPI_WaitTxFifoEmpty(LPSPI_Type *base)

Wait for tx FIFO to be empty.

This function wait the tx fifo empty

Parameters

- base – LPSPI peripheral address.

Returns

true for the tx FIFO is ready, false is not.

void LPSPI_DriverIRQHandler(uint32_t instance)

LPSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for LPSPI.

Parameters

- instance – LPSPI instance.

FSL_LPSPI_DRIVER_VERSION

LPSPI driver version.

Status for the LPSPI driver.

Values:

enumerator kStatus_LPSPI_Busy

LPSPI transfer is busy.

enumerator kStatus_LPSPI_Error

LPSPI driver error.

enumerator kStatus_LPSPI_Idle

LPSPI is idle.

enumerator kStatus_LPSPI_OutOfRange

LPSPI transfer out Of range.

enumerator kStatus_LPSPI_Timeout

LPSPI timeout polling status flags.

enum *_lpspi_flags*

LPSPI status flags in SPIx_SR register.

Values:

enumerator kLPSPI_TxDataRequestFlag

Transmit data flag

enumerator kLPSPI_RxDataReadyFlag

Receive data flag

enumerator kLPSPI_WordCompleteFlag
Word Complete flag

enumerator kLPSPI_FrameCompleteFlag
Frame Complete flag

enumerator kLPSPI_TransferCompleteFlag
Transfer Complete flag

enumerator kLPSPI_TransmitErrorFlag
Transmit Error flag (FIFO underrun)

enumerator kLPSPI_ReceiveErrorFlag
Receive Error flag (FIFO overrun)

enumerator kLPSPI_DataMatchFlag
Data Match flag

enumerator kLPSPI_ModuleBusyFlag
Module Busy flag

enumerator kLPSPI_AllStatusFlag
Used for clearing all w1c status flags

enum _lpspi_interrupt_enable
LPSPI interrupt source.

Values:

enumerator kLPSPI_TxInterruptEnable
Transmit data interrupt enable

enumerator kLPSPI_RxInterruptEnable
Receive data interrupt enable

enumerator kLPSPI_WordCompleteInterruptEnable
Word complete interrupt enable

enumerator kLPSPI_FrameCompleteInterruptEnable
Frame complete interrupt enable

enumerator kLPSPI_TransferCompleteInterruptEnable
Transfer complete interrupt enable

enumerator kLPSPI_TransmitErrorInterruptEnable
Transmit error interrupt enable(FIFO underrun)

enumerator kLPSPI_ReceiveErrorInterruptEnable
Receive Error interrupt enable (FIFO overrun)

enumerator kLPSPI_DataMatchInterruptEnable
Data Match interrupt enable

enumerator kLPSPI_AllInterruptEnable
All above interrupts enable.

enum _lpspi_dma_enable
LPSPI DMA source.

Values:

enumerator kLPSPI_TxDmaEnable
Transmit data DMA enable

enumerator kLPSPI_RxDmaEnable

Receive data DMA enable

enum _lpspi_master_slave_mode

LPSPI master or slave mode configuration.

Values:

enumerator kLPSPI_Master

LPSPI peripheral operates in master mode.

enumerator kLPSPI_Slave

LPSPI peripheral operates in slave mode.

enum _lpspi_which_pcs_config

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

Values:

enumerator kLPSPI_Pcs0

PCS[0]

enumerator kLPSPI_Pcs1

PCS[1]

enumerator kLPSPI_Pcs2

PCS[2]

enumerator kLPSPI_Pcs3

PCS[3]

enum _lpspi_pcs_polarity_config

LPSPI Peripheral Chip Select (PCS) Polarity configuration.

Values:

enumerator kLPSPI_PcsActiveHigh

PCS Active High (idles low)

enumerator kLPSPI_PcsActiveLow

PCS Active Low (idles high)

enum _lpspi_pcs_polarity

LPSPI Peripheral Chip Select (PCS) Polarity.

Values:

enumerator kLPSPI_Pcs0ActiveLow

Pcs0 Active Low (idles high).

enumerator kLPSPI_Pcs1ActiveLow

Pcs1 Active Low (idles high).

enumerator kLPSPI_Pcs2ActiveLow

Pcs2 Active Low (idles high).

enumerator kLPSPI_Pcs3ActiveLow

Pcs3 Active Low (idles high).

enumerator kLPSPI_PcsAllActiveLow

Pcs0 to Pcs5 Active Low (idles high).

enum `_lpspi_clock_polarity`

LPSPI clock polarity configuration.

Values:

enumerator `kLPSPi_ClockPolarityActiveHigh`

CPOL=0. Active-high LPSPi clock (idles low)

enumerator `kLPSPi_ClockPolarityActiveLow`

CPOL=1. Active-low LPSPi clock (idles high)

enum `_lpspi_clock_phase`

LPSPi clock phase configuration.

Values:

enumerator `kLPSPi_ClockPhaseFirstEdge`

CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

enumerator `kLPSPi_ClockPhaseSecondEdge`

CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

enum `_lpspi_shift_direction`

LPSPi data shifter direction options.

Values:

enumerator `kLPSPi_MsbFirst`

Data transfers start with most significant bit.

enumerator `kLPSPi_LsbFirst`

Data transfers start with least significant bit.

enum `_lpspi_host_request_select`

LPSPi Host Request select configuration.

Values:

enumerator `kLPSPi_HostReqExtPin`

Host Request is an ext pin.

enumerator `kLPSPi_HostReqInternalTrigger`

Host Request is an internal trigger.

enum `_lpspi_match_config`

LPSPi Match configuration options.

Values:

enumerator `kLPSPi_MatchDisabled`

LPSPi Match Disabled.

enumerator `kLPSPi_1stWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSPi_AnyWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSPi_1stWordEqualsM0and2ndWordEqualsM1`

LPSPi Match Enabled.

enumerator `kLPSPi_AnyWordEqualsM0andNxtWordEqualsM1`

LPSPi Match Enabled.

enumerator kLPSI_1stWordAndM1EqualsM0andM1
LPSPi Match Enabled.

enumerator kLPSI_AnyWordAndM1EqualsM0andM1
LPSPi Match Enabled.

enum _lpspi_pin_config
LPSPi pin (SDO and SDI) configuration.

Values:

enumerator kLPSPI_SdiInSdoOut
LPSPi SDI input, SDO output.

enumerator kLPSPI_SdiInSdiOut
LPSPi SDI input, SDI output.

enumerator kLPSPI_SdoInSdoOut
LPSPi SDO input, SDO output.

enumerator kLPSPI_SdoInSdiOut
LPSPi SDO input, SDI output.

enum _lpspi_data_out_config
LPSPi data output configuration.

Values:

enumerator kLpspiDataOutRetained
Data out retains last value when chip select is de-asserted

enumerator kLpspiDataOutTristate
Data out is tristated when chip select is de-asserted

enum _lpspi_transfer_width
LPSPi transfer width configuration.

Values:

enumerator kLPSPI_SingleBitXfer
1-bit shift at a time, data out on SDO, in on SDI (normal mode)

enumerator kLPSPI_TwoBitXfer
2-bits shift out on SDO/SDI and in on SDO/SDI

enumerator kLPSPI_FourBitXfer
4-bits shift out on SDO/SDI/PCS[3:2] and in on SDO/SDI/PCS[3:2]

enum _lpspi_delay_type
LPSPi delay type selection.

Values:

enumerator kLPSPI_PcsToSck
PCS-to-SCK delay.

enumerator kLPSPI_LastSckToPcs
Last SCK edge to PCS delay.

enumerator kLPSPI_BetweenTransfer
Delay between transfers.

enum _lpspi_transfer_config_flag_for_master

Use this enumeration for LPSPi master transfer configFlags.

Values:

enumerator kLPSPi_MasterPcs0

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS0 signal

enumerator kLPSPi_MasterPcs1

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS1 signal

enumerator kLPSPi_MasterPcs2

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS2 signal

enumerator kLPSPi_MasterPcs3

LPSPi master PCS shift macro , internal used. LPSPi master transfer use PCS3 signal

enumerator kLPSPi_MasterPcsContinuous

Is PCS signal continuous

enumerator kLPSPi_MasterByteSwap

Is master swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

- i. If you set bitPerFrame = 8 , no matter the kLPSPi_MasterByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.

enum _lpspi_transfer_config_flag_for_slave

Use this enumeration for LPSPi slave transfer configFlags.

Values:

enumerator kLPSPi_SlavePcs0

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS0 signal

enumerator kLPSPi_SlavePcs1

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS1 signal

enumerator kLPSPi_SlavePcs2

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS2 signal

enumerator kLPSPi_SlavePcs3

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS3 signal

enumerator kLPSPi_SlaveByteSwap

Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

- i. If you set bitPerFrame = 8 , no matter the kLPSPi_SlaveByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.

enum `_lpspi_transfer_state`

LPSPI transfer state, which is used for LPSPI transactional API state machine.

Values:

enumerator `kLPSPI_Idle`

Nothing in the transmitter/receiver.

enumerator `kLPSPI_Busy`

Transfer queue is not finished.

enumerator `kLPSPI_Error`

Transfer error.

typedef enum `_lpspi_master_slave_mode` `lpspi_master_slave_mode_t`

LPSPI master or slave mode configuration.

typedef enum `_lpspi_which_pcs_config` `lpspi_which_pcs_t`

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

typedef enum `_lpspi_pcs_polarity_config` `lpspi_pcs_polarity_config_t`

LPSPI Peripheral Chip Select (PCS) Polarity configuration.

typedef enum `_lpspi_clock_polarity` `lpspi_clock_polarity_t`

LPSPI clock polarity configuration.

typedef enum `_lpspi_clock_phase` `lpspi_clock_phase_t`

LPSPI clock phase configuration.

typedef enum `_lpspi_shift_direction` `lpspi_shift_direction_t`

LPSPI data shifter direction options.

typedef enum `_lpspi_host_request_select` `lpspi_host_request_select_t`

LPSPI Host Request select configuration.

typedef enum `_lpspi_match_config` `lpspi_match_config_t`

LPSPI Match configuration options.

typedef enum `_lpspi_pin_config` `lpspi_pin_config_t`

LPSPI pin (SDO and SDI) configuration.

typedef enum `_lpspi_data_out_config` `lpspi_data_out_config_t`

LPSPI data output configuration.

typedef enum `_lpspi_transfer_width` `lpspi_transfer_width_t`

LPSPI transfer width configuration.

typedef enum `_lpspi_delay_type` `lpspi_delay_type_t`

LPSPI delay type selection.

typedef struct `_lpspi_master_config` `lpspi_master_config_t`

LPSPI master configuration structure.

typedef struct `_lpspi_slave_config` `lpspi_slave_config_t`

LPSPI slave configuration structure.

typedef struct `_lpspi_master_handle` `lpspi_master_handle_t`

Forward declaration of the `_lpspi_master_handle` typedefs.

typedef struct `_lpspi_slave_handle` `lpspi_slave_handle_t`

Forward declaration of the `_lpspi_slave_handle` typedefs.


```
typedef void (*lpspi_master_transfer_callback_t)(LPSPI_Type *base, lpspi_master_handle_t
*handle, status_t status, void *userData)
```

Master completion callback function pointer type.

Param base

LPSPI peripheral address.

Param handle

Pointer to the handle for the LPSPI master.

Param status

Success or error code describing whether the transfer is completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
typedef void (*lpspi_slave_transfer_callback_t)(LPSPI_Type *base, lpspi_slave_handle_t *handle,
status_t status, void *userData)
```

Slave completion callback function pointer type.

Param base

LPSPI peripheral address.

Param handle

Pointer to the handle for the LPSPI slave.

Param status

Success or error code describing whether the transfer is completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
typedef struct _lpspi_transfer lpspi_transfer_t
```

LPSPI master/slave transfer structure.

```
volatile uint8_t g_lpspiDummyData[]
```

Global variable for dummy data value setting.

```
LPSPI_DUMMY_DATA
```

LPSPI dummy data if no Tx data.

Dummy data used for tx if there is not txData.

```
SPI_RETRY_TIMES
```

Retry times for waiting flag.

```
LPSPI_MASTER_PCS_SHIFT
```

LPSPI master PCS shift macro , internal used.

```
LPSPI_MASTER_PCS_MASK
```

LPSPI master PCS shift macro , internal used.

```
LPSPI_SLAVE_PCS_SHIFT
```

LPSPI slave PCS shift macro , internal used.

```
LPSPI_SLAVE_PCS_MASK
```

LPSPI slave PCS shift macro , internal used.

```
struct _lpspi_master_config
```

#include <fsl_lpspi.h> LPSPI master configuration structure.

Public Members

uint32_t baudRate

Baud Rate for LPSPI.

uint32_t bitsPerFrame

Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol

Clock polarity.

lpspi_clock_phase_t cpha

Clock phase.

lpspi_shift_direction_t direction

MSB or LSB data shift direction.

uint32_t pcsToSckDelayInNanoSec

PCS to SCK delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

uint32_t lastSckToPcsDelayInNanoSec

Last SCK to PCS delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

uint32_t betweenTransferDelayInNanoSec

After the SCK delay time with nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

lpspi_which_pcs_t whichPcs

Desired Peripheral Chip Select (PCS).

lpspi_pcs_polarity_config_t pcsActiveHighOrLow

Desired PCS active high or low

lpspi_pin_config_t pinCfg

Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

bool enableInputDelay

Enable master to sample the input data on a delayed SCK. This can help improve slave setup time. Refer to device data sheet for specific time length.

struct __lpspi_slave_config

#include <fsl_lpspi.h> LPSPI slave configuration structure.

Public Members

uint32_t bitsPerFrame

Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol

Clock polarity.

lpspi_clock_phase_t cpha

Clock phase.

lpspi_shift_direction_t direction

MSB or LSB data shift direction.

lpspi_which_pcs_t whichPcs

Desired Peripheral Chip Select (pcs)

lpspi_pcs_polarity_config_t pcsActiveHighOrLow

Desired PCS active high or low

lpspi_pin_config_t pinCfg

Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

struct *_lpspi_transfer*

#include <fsl_lpspi.h> LPSPI master/slave transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

volatile size_t dataSize

Transfer bytes.

uint32_t configFlags

Transfer transfer configuration flags. Set from *_lpspi_transfer_config_flag_for_master* if the transfer is used for master or *_lpspi_transfer_config_flag_for_slave* enumeration if the transfer is used for slave.

struct *_lpspi_master_handle*

#include <fsl_lpspi.h> LPSPI master transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous

Is PCS continuous in transfer.

volatile bool writeTcrInIsr

A flag that whether should write TCR in ISR.

volatile bool isByteSwap

A flag that whether should byte swap.

volatile bool isTxMask

A flag that whether TCR[TXMSK] is set.

volatile uint16_t bytesPerFrame

Number of bytes in each frame

volatile uint16_t frameSize

Backup of TCR[FRAMESZ]

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

`volatile uint8_t bytesEachWrite`
Bytes for each write TDR.

`volatile uint8_t bytesEachRead`
Bytes for each read RDR.

`const uint8_t *volatile txData`
Send buffer.

`uint8_t *volatile rxData`
Receive buffer.

`volatile size_t txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`uint32_t txBuffIfNull`
Used if the txData is NULL.

`volatile uint8_t state`
LPSPI transfer state , `_lpspi_transfer_state`.

`lpspi_master_transfer_callback_t callback`
Completion callback.

`void *userData`
Callback user data.

`struct _lpspi_slave_handle`
#include <fsl_lpspi.h> LPSPI slave transfer handle structure used for transactional API.

Public Members

`volatile bool isByteSwap`
A flag that whether should byte swap.

`volatile uint8_t fifoSize`
FIFO dataSize.

`volatile uint8_t rxWatermark`
Rx watermark.

`volatile uint8_t bytesEachWrite`
Bytes for each write TDR.

`volatile uint8_t bytesEachRead`
Bytes for each read RDR.

`const uint8_t *volatile txData`
Send buffer.

`uint8_t *volatile rxData`
Receive buffer.

`volatile size_t txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`volatile uint8_t state`
LPSPI transfer state , `_lpspi_transfer_state`.

`volatile uint32_t errorCount`
Error count for slave transfer.

`lpspi_slave_transfer_callback_t callback`
Completion callback.

`void *userData`
Callback user data.

2.40 LPSPI eDMA Driver

`FSL_LPSPi_EDMA_DRIVER_VERSION`
LPSPI EDMA driver version.

`DMA_MAX_TRANSFER_COUNT`
DMA max transfer size.

`typedef struct _lpspi_master_edma_handle lpspi_master_edma_handle_t`
Forward declaration of the `_lpspi_master_edma_handle` typedefs.

`typedef struct _lpspi_slave_edma_handle lpspi_slave_edma_handle_t`
Forward declaration of the `_lpspi_slave_edma_handle` typedefs.

`typedef void (*lpspi_master_edma_transfer_callback_t)(LPSPI_Type *base,
lpspi_master_edma_handle_t *handle, status_t status, void *userData)`
Completion callback function pointer type.

Param base
LPSPI peripheral base address.

Param handle
Pointer to the handle for the LPSPI master.

Param status
Success or error code describing whether the transfer completed.

Param userData
Arbitrary pointer-dataSized value passed from the application.

```
typedef void (*lpspi_slave_edma_transfer_callback_t)(LPSPI_Type *base,  
lpspi_slave_edma_handle_t *handle, status_t status, void *userData)
```

Completion callback function pointer type.

Param base

LPSPI peripheral base address.

Param handle

Pointer to the handle for the LPSPI slave.

Param status

Success or error code describing whether the transfer completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
void LPSPI_MasterTransferCreateHandleEDMA(LPSPI_Type *base, lpspi_master_edma_handle_t  
*handle, lpspi_master_edma_transfer_callback_t  
callback, void *userData, edma_handle_t  
*edmaRxRegToRxDataHandle, edma_handle_t  
*edmaTxDataToTxRegHandle)
```

Initializes the LPSPI master eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that the LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx are the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Tx DMAMUX source for edmaRxRegToRxDataHandle.

Parameters

- base – LPSPI peripheral base address.
- handle – LPSPI handle pointer to lpspi_master_edma_handle_t.
- callback – LPSPI callback.
- userData – callback function parameter.
- edmaRxRegToRxDataHandle – edmaRxRegToRxDataHandle pointer to edma_handle_t.
- edmaTxDataToTxRegHandle – edmaTxDataToTxRegHandle pointer to edma_handle_t.

```
status_t LPSPI_MasterTransferEDMA(LPSPI_Type *base, lpspi_master_edma_handle_t *handle,  
lpspi_transfer_t *transfer)
```

LPSPI master transfer data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.

- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_MasterTransferPrepareEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, uint32_t configFlags)

LPSPI master config transfer parameter while using eDMA.

This function is preparing to transfer data using eDMA, work with LPSPI_MasterTransferEDMALite.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- configFlags – transfer configuration flags. `_lpspi_transfer_config_flag_for_master`.

Return values

- `kStatus_Success` – Execution successfully.
- `kStatus_LPSPI_Busy` – The LPSPI device is busy.

Returns

Indicates whether LPSPI master transfer was successful or not.

`status_t` LPSPI_MasterTransferEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA without configs.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: This API is only for transfer through DMA without configuration. Before calling this API, you must call LPSPI_MasterTransferPrepareEDMALite to configure it once. The transfer data size should be an integer multiple of `bytesPerFrame` if `bytesPerFrame` is less than or equal to 4. For `bytesPerFrame` greater than 4: The transfer data size should be equal to `bytesPerFrame` if the `bytesPerFrame` is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of `bytesPerFrame`.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- transfer – pointer to `lpspi_transfer_t` structure, config field is not used.

Return values

- `kStatus_Success` – Execution successfully.
- `kStatus_LPSPI_Busy` – The LPSPI device is busy.
- `kStatus_InvalidArgument` – The transfer structure is invalid.

Returns

Indicates whether LPSPI master transfer was successful or not.

`void` LPSPI_MasterTransferAbortEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle)

LPSPI master aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.

status_t LPSPI_MasterTransferGetCountEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *size_t* *count)

Gets the master eDMA transfer remaining bytes.

This function gets the master eDMA transfer remaining bytes.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the EDMA transaction.

Returns

status of *status_t*.

void LPSPI_SlaveTransferCreateHandleEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *lpspi_slave_edma_transfer_callback_t* callback, *void* *userData, *edma_handle_t* *edmaRxRegToRxDataHandle, *edma_handle_t* *edmaTxDataToTxRegHandle)

Initializes the LPSPI slave eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx as the same source) DMA request source.

(1) For a separated DMA request source, enable and set the Rx DMAMUX source for `edmaRxRegToRxDataHandle` and Tx DMAMUX source for `edmaTxDataToTxRegHandle`. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for `edmaRxRegToRxDataHandle`.

Parameters

- base – LPSPI peripheral base address.
- handle – LPSPI handle pointer to `lpspi_slave_edma_handle_t`.
- callback – LPSPI callback.
- userData – callback function parameter.
- `edmaRxRegToRxDataHandle` – `edmaRxRegToRxDataHandle` pointer to `edma_handle_t`.
- `edmaTxDataToTxRegHandle` – `edmaTxDataToTxRegHandle` pointer to `edma_handle_t`.

status_t LPSPI_SlaveTransferEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfers data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which return right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of `bytesPerFrame` if `bytesPerFrame` is less than or equal to 4. For `bytesPerFrame` greater than 4: The transfer data size

should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`void LPSPI_SlaveTransferAbortEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle)`
LPSPI slave aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.

`status_t LPSPI_SlaveTransferGetCountEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, size_t *count)`

Gets the slave eDMA transfer remaining bytes.

This function gets the slave eDMA transfer remaining bytes.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the eDMA transaction.

Returns

status of `status_t`.

`struct _lpspi_master_edma_handle`

#include <fsl_lpspi_edma.h> LPSPI master eDMA transfer handle structure used for transactional API.

Public Members

`volatile bool isPcsContinuous`

Is PCS continuous in transfer.

`volatile bool isByteSwap`

A flag that whether should byte swap.

`volatile uint8_t fifoSize`

FIFO dataSize.

`volatile uint8_t rxWatermark`

Rx watermark.

`volatile uint8_t bytesEachWrite`

Bytes for each write TDR.

`volatile uint8_t bytesEachRead`
Bytes for each read RDR.

`volatile uint8_t bytesLastRead`
Bytes for last read RDR.

`volatile bool isThereExtraRxBytes`
Is there extra RX byte.

`const uint8_t *volatile txData`
Send buffer.

`uint8_t *volatile rxData`
Receive buffer.

`volatile size_t txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`edma_tcd_t *lastTimeTCD`
Pointer to the lastTime TCD

`bool isMultiDMATransmit`
Is there multi DMA transmit

`volatile uint8_t dmaTransmitTime`
DMA Transfer times.

`uint32_t lastTimeDataBytes`
DMA transmit last Time data Bytes

`uint32_t dataBytesEveryTime`
Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

`edma_transfer_config_t transferConfigRx`
Config of DMA rx channel.

`edma_transfer_config_t transferConfigTx`
Config of DMA tx channel.

`uint32_t txBuffIfNull`
Used if there is not txData for DMA purpose.

`uint32_t rxBuffIfNull`
Used if there is not rxData for DMA purpose.

`uint32_t transmitCommand`
Used to write TCR for DMA purpose.

`volatile uint8_t state`
LPSPI transfer state , `_lpspi_transfer_state`.

```

uint8_t nbytes
    eDMA minor byte transfer count initially configured.

lpspi_master_edma_transfer_callback_t callback
    Completion callback.

void *userData
    Callback user data.

edma_handle_t *edmaRxRegToRxDataHandle
    edma_handle_t handle point used for RxReg to RxData buff

edma_handle_t *edmaTxDataToTxRegHandle
    edma_handle_t handle point used for TxData to TxReg buff

edma_tcd_t lpspiSoftwareTCD[3]
    SoftwareTCD, internal used

```

struct `_lpspi_slave_edma_handle`
#include <fsl_lpspi_edma.h> LPSPI slave eDMA transfer handle structure used for transactional API.

Public Members

```

volatile bool isByteSwap
    A flag that whether should byte swap.

volatile uint8_t fifoSize
    FIFO dataSize.

volatile uint8_t rxWatermark
    Rx watermark.

volatile uint8_t bytesEachWrite
    Bytes for each write TDR.

volatile uint8_t bytesEachRead
    Bytes for each read RDR.

volatile uint8_t bytesLastRead
    Bytes for last read RDR.

volatile bool isThereExtraRxBytes
    Is there extra RX byte.

uint8_t nbytes
    eDMA minor byte transfer count initially configured.

const uint8_t *volatile txData
    Send buffer.

uint8_t *volatile rxData
    Receive buffer.

volatile size_t txRemainingByteCount
    Number of bytes remaining to send.

volatile size_t rxRemainingByteCount
    Number of bytes remaining to receive.

```

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`uint32_t txBuffIfNull`
Used if there is not txData for DMA purpose.

`uint32_t rxBuffIfNull`
Used if there is not rxData for DMA purpose.

`volatile uint8_t state`
LPSPI transfer state.

`uint32_t errorCount`
Error count for slave transfer.

`lpspi_slave_edma_transfer_callback_t callback`
Completion callback.

`void *userData`
Callback user data.

`edma_handle_t *edmaRxRegToRxDataHandle`
edma_handle_t handle point used for RxReg to RxData buff

`edma_handle_t *edmaTxDataToTxRegHandle`
edma_handle_t handle point used for TxData to TxReg

`edma_tcd_t lpspiSoftwareTCD[2]`
SoftwareTCD, internal used

2.41 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.42 LPUART Driver

`static inline void LPUART_SoftwareReset(LPUART_Type *base)`

Resets the LPUART using software.

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

- `base` – LPUART peripheral base address.

`status_t LPUART_Init(LPUART_Type *base, const lpuart_config_t *config, uint32_t srcClock_Hz)`

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the `LPUART_GetDefaultConfig()` function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

status_t LPUART_Deinit(LPUART_Type *base)

Deinitializes a LPUART instance.

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

- base – LPUART peripheral base address.

Return values

- kStatus_Success – Deinit is success.
- kStatus_LPUART_Timeout – Timeout during deinit.

void LPUART_GetDefaultConfig(*lpuart_config_t* *config)

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus_Success – Set baudrate succeeded.

```
void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)
```

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, false to disable.

```
static inline void LPUART_SetMatchAddress(LPUART_Type *base, uint16_t address1, uint16_t  
                                         address2)
```

Set the LPUART address.

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – LPUART peripheral base address.
- address1 – LPUART slave address1.
- address2 – LPUART slave address2.

```
static inline void LPUART_EnableMatchAddress(LPUART_Type *base, bool match1, bool  
                                             match2)
```

Enable the LPUART match address feature.

Parameters

- base – LPUART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

```
static inline void LPUART_SetRxFifoWatermark(LPUART_Type *base, uint8_t water)
```

Sets the rx FIFO watermark.

Parameters

- base – LPUART peripheral base address.

- water – Rx FIFO watermark.

static inline void LPUART_SetTxFifoWatermark(LPUART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Tx FIFO watermark.

static inline void LPUART_TransferEnable16Bit(*lpuart_handle_t* *handle, bool enable)

Sets the LPUART using 16bit transmit, only for 9bit or 10bit mode.

This function Enable 16bit Data transmit in *lpuart_handle_t*.

Parameters

- handle – LPUART handle pointer.
- enable – true to enable, false to disable.

uint32_t LPUART_GetStatusFlags(LPUART_Type *base)

Gets LPUART status flags.

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators *_lpuart_flags*. To check for a specific status, compare the return value with enumerators in the *_lpuart_flags*. For example, to check whether the TX is empty:

```
if (kLPUART_TxDataRegEmptyFlag & LPUART_GetStatusFlags(LPUART1))
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART status flags which are Ored by the enumerators in the *_lpuart_flags*.

status_t LPUART_ClearStatusFlags(LPUART_Type *base, uint32_t mask)

Clears status flags with a provided mask.

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only cleared or set by hardware are: *kLPUART_TxDataRegEmptyFlag*, *kLPUART_TransmissionCompleteFlag*, *kLPUART_RxDataRegFullFlag*, *kLPUART_RxActiveFlag*, *kLPUART_NoiseErrorFlag*, *kLPUART_ParityErrorFlag*, *kLPUART_TxFifoEmptyFlag*, *kLPUART_RxFifoEmptyFlag* Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- base – LPUART peripheral base address.
- mask – the status flags to be cleared. The user can use the enumerators in the *_lpuart_status_flag_t* to do the OR operation and get the mask.

Return values

- *kStatus_LPUART_FlagCannotClearManually* – The flag can't be cleared by this function but it is cleared automatically by hardware.
- *kStatus_Success* – Status in the mask are cleared.

Returns

0 succeed, others failed.

`void LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)`

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the `_lpuart_interrupt_enable`. This examples shows how to enable TX empty interrupt and RX full interrupt:

```
LPUART_EnableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – LPUART peripheral base address.
- `mask` – The interrupts to enable. Logical OR of `_lpuart_interrupt_enable`.

`void LPUART_DisableInterrupts(LPUART_Type *base, uint32_t mask)`

Disables LPUART interrupts according to a provided mask.

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpuart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
LPUART_DisableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – LPUART peripheral base address.
- `mask` – The interrupts to disable. Logical OR of `_lpuart_interrupt_enable`.

`uint32_t LPUART_GetEnabledInterrupts(LPUART_Type *base)`

Gets enabled LPUART interrupts.

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_lpuart_interrupt_enable`. To check a specific interrupt enable status, compare the return value with enumerators in `_lpuart_interrupt_enable`. For example, to check whether the TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);

if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

- `base` – LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in `_lpuart_interrupt_enable`.

`static inline uintptr_t LPUART_GetDataRegisterAddress(LPUART_Type *base)`

Gets the LPUART data register address.

This function returns the LPUART data register address, which is mainly used by the DMA/eDMA.

Parameters

- `base` – LPUART peripheral base address.

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

static inline void LPUART_EnableTxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter DMA request.

This function enables or disables the transmit data register empty flag, STAT[TDRE], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver DMA.

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

uint32_t LPUART_GetInstance(LPUART_Type *base)

Get the LPUART instance from peripheral base address.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART instance.

static inline void LPUART_EnableTx(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter.

This function enables or disables the LPUART transmitter.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRx(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver.

This function enables or disables the LPUART receiver.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_WriteByte(LPUART_Type *base, uint8_t data)

Writes to the transmitter register.

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

- base – LPUART peripheral base address.
- data – Data write to the TX register.

static inline uint8_t LPUART_ReadByte(LPUART_Type *base)

Reads the receiver register.

This function reads data from the receiver register directly. The upper layer must ensure that the receiver register is full or that the RX FIFO has data before calling this function.

Parameters

- base – LPUART peripheral base address.

Returns

Data read from data register.

static inline uint8_t LPUART_GetRxFifoCount(LPUART_Type *base)

Gets the rx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

rx FIFO data count.

static inline uint8_t LPUART_GetTxFifoCount(LPUART_Type *base)

Gets the tx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

tx FIFO data count.

void LPUART_SendAddress(LPUART_Type *base, uint8_t address)

Transmit an address frame in 9-bit data mode.

Parameters

- base – LPUART peripheral base address.
- address – LPUART slave address.

status_t LPUART_WriteBlocking(LPUART_Type *base, const uint8_t *data, size_t length)

Writes to the transmitter register using a blocking method.

This function polls the transmitter register, first waits for the register to be empty or TX FIFO to have room, and writes data to the transmitter buffer, then waits for the data to be sent out to the bus.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_WriteBlocking16bit(LPUART_Type *base, const uint16_t *data, size_t length)

Writes to the transmitter register using a blocking method in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer. Please make sure only 10bit of data is valid and other bits are 0.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_ReadBlocking(LPUART_Type *base, uint8_t *data, size_t length)

Reads the receiver data register using a blocking method.

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the TX register.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.
- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t LPUART_ReadBlocking16bit(LPUART_Type *base, uint16_t *data, size_t length)

Reads the receiver data register in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data by 16bit, only 10bit is valid.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.

- `kStatus_LPUART_ParityError` – Parity error happened while receiving data.
- `kStatus_LPUART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

```
void LPUART__TransferCreateHandle(LPUART_Type *base, lpuart_handle_t *handle,  
                                lpuart_transfer_callback_t callback, void *userData)
```

Initializes the LPUART handle.

This function initializes the LPUART handle, which can be used for other LPUART transactional APIs. Usually, for a specified LPUART instance, call this API once to get the initialized handle.

The LPUART driver supports the “background” receiving, which means that user can set up an RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn’t call the `LPUART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as `ringBuffer`.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `callback` – Callback function.
- `userData` – User data.

```
status_t LPUART__TransferSendNonBlocking(LPUART_Type *base, lpuart_handle_t *handle,  
                                         lpuart_transfer_t *xfer)
```

Transmits a buffer of data using the interrupt method.

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the `kStatus_LPUART_TxIdle` as status parameter.

Note: The `kStatus_LPUART_TxIdle` is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the TX, check the `kLPUART_TransmissionCompleteFlag` to ensure that the transmit is finished.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART transfer structure, see `lpuart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_LPUART_TxBusy` – Previous transmission still not finished, data not all written to the TX register.
- `kStatus_InvalidArgument` – Invalid argument.

```
void LPUART__TransferStartRingBuffer(LPUART_Type *base, lpuart_handle_t *handle, uint8_t  
                                     *ringBuffer, size_t ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the `UART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `ringBuffer` – Start address of ring buffer for background receiving. Pass `NULL` to disable the ring buffer.
- `ringBufferSize` – size of the ring buffer.

`void LPUART__TransferStopRingBuffer(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`size_t LPUART__TransferGetRxRingBufferLength(LPUART_Type *base, lpuart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

Returns

Length of received data in RX ring buffer.

`void LPUART__TransferAbortSend(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are not sent out.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t LPUART__TransferGetSendCount(LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by an interrupt method.

Parameters

- `base` – LPUART peripheral base address.

- `handle` – LPUART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`status_t` LPUART_TransferReceiveNonBlocking(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_t* *xfer, `size_t` *receivedBytes)

Receives a buffer of data using the interrupt method.

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter `kStatus_UART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to `xfer->data`, which returns with the parameter `receivedBytes` set to 5. For the remaining 5 bytes, the newly arrived data is saved from `xfer->data[5]`. When 5 bytes are received, the LPUART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART transfer structure, see `uart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into the transmit queue.
- `kStatus_LPUART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void` LPUART_TransferAbortReceive(LPUART_Type *base, *lpuart_handle_t* *handle)

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to find out how many bytes not received yet.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t` LPUART_TransferGetReceiveCount(LPUART_Type *base, *lpuart_handle_t* *handle, `uint32_t` *count)

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – LPUART peripheral base address.

- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

void LPUART_TransferHandleIRQ(LPUART_Type *base, void *irqHandle)
LPUART IRQ handle function.

This function handles the LPUART transmit and receive IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_TransferHandleErrorIRQ(LPUART_Type *base, void *irqHandle)
LPUART Error IRQ handle function.

This function handles the LPUART error IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_DriverIRQHandler(uint32_t instance)
LPUART driver IRQ handler common entry.

This function provides the common IRQ request entry for LPUART.

Parameters

- instance – LPUART instance.

FSL_LPUART_DRIVER_VERSION
LPUART driver version.

Error codes for the LPUART driver.

Values:

enumerator kStatus_LPUART_TxBusy
TX busy

enumerator kStatus_LPUART_RxBusy
RX busy

enumerator kStatus_LPUART_TxIdle
LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle
LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge
TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge
RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually

Some flag can't manually clear

enumerator kStatus_LPUART_Error

Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun

LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun

LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError

LPUART noise error.

enumerator kStatus_LPUART_FramingError

LPUART framing error.

enumerator kStatus_LPUART_ParityError

LPUART parity error.

enumerator kStatus_LPUART_BaudrateNotSupport

Baudrate is not support in current clock source

enumerator kStatus_LPUART_IdleLineDetected

IDLE flag.

enumerator kStatus_LPUART_Timeout

LPUART times out.

enum _lpuart_parity_mode

LPUART parity mode.

Values:

enumerator kLPUART_ParityDisabled

Parity disabled

enumerator kLPUART_ParityEven

Parity enabled, type even, bit setting: PE | PT = 10

enumerator kLPUART_ParityOdd

Parity enabled, type odd, bit setting: PE | PT = 11

enum _lpuart_data_bits

LPUART data bits count.

Values:

enumerator kLPUART_EightDataBits

Eight data bit

enumerator kLPUART_SevenDataBits

Seven data bit

enum _lpuart_stop_bit_count

LPUART stop bit count.

Values:

enumerator kLPUART_OneStopBit

One stop bit

enumerator kLPUART_TwoStopBit

Two stop bits

enum _lpuart_transmit_cts_source

LPUART transmit CTS source.

Values:

enumerator kLPUART_CtsSourcePin

CTS resource is the LPUART_CTS pin.

enumerator kLPUART_CtsSourceMatchResult

CTS resource is the match result.

enum _lpuart_transmit_cts_config

LPUART transmit CTS configure.

Values:

enumerator kLPUART_CtsSampleAtStart

CTS input is sampled at the start of each character.

enumerator kLPUART_CtsSampleAtIdle

CTS input is sampled when the transmitter is idle

enum _lpuart_idle_type_select

LPUART idle flag type defines when the receiver starts counting.

Values:

enumerator kLPUART_IdleTypeStartBit

Start counting after a valid start bit.

enumerator kLPUART_IdleTypeStopBit

Start counting after a stop bit.

enum _lpuart_idle_config

LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

Values:

enumerator kLPUART_IdleCharacter1

the number of idle characters.

enumerator kLPUART_IdleCharacter2

the number of idle characters.

enumerator kLPUART_IdleCharacter4

the number of idle characters.

enumerator kLPUART_IdleCharacter8

the number of idle characters.

enumerator kLPUART_IdleCharacter16

the number of idle characters.

enumerator kLPUART_IdleCharacter32

the number of idle characters.

enumerator kLPUART_IdleCharacter64

the number of idle characters.

enumerator kLPUART_IdleCharacter128
the number of idle characters.

enum _lpuart_interrupt_enable

LPUART interrupt configuration structure, default settings all disabled.

This structure contains the settings for all LPUART interrupt configurations.

Values:

enumerator kLPUART_LinBreakInterruptEnable
LIN break detect. bit 7

enumerator kLPUART_RxActiveEdgeInterruptEnable
Receive Active Edge. bit 6

enumerator kLPUART_TxDataRegEmptyInterruptEnable
Transmit data register empty. bit 23

enumerator kLPUART_TransmissionCompleteInterruptEnable
Transmission complete. bit 22

enumerator kLPUART_RxDataRegFullInterruptEnable
Receiver data register full. bit 21

enumerator kLPUART_IdleLineInterruptEnable
Idle line. bit 20

enumerator kLPUART_RxOverrunInterruptEnable
Receiver Overrun. bit 27

enumerator kLPUART_NoiseErrorInterruptEnable
Noise error flag. bit 26

enumerator kLPUART_FramingErrorInterruptEnable
Framing error flag. bit 25

enumerator kLPUART_ParityErrorInterruptEnable
Parity error flag. bit 24

enumerator kLPUART_Match1InterruptEnable
Parity error flag. bit 15

enumerator kLPUART_Match2InterruptEnable
Parity error flag. bit 14

enumerator kLPUART_TxFifoOverflowInterruptEnable
Transmit FIFO Overflow. bit 9

enumerator kLPUART_RxFifoUnderflowInterruptEnable
Receive FIFO Underflow. bit 8

enumerator kLPUART_AllInterruptEnable

enum _lpuart_flags

LPUART status flags.

This provides constants for the LPUART status flags for use in the LPUART functions.

Values:

enumerator kLPUART_TxDataRegEmptyFlag
Transmit data register empty flag, sets when transmit buffer is empty. bit 23

enumerator `kLPUART_TransmissionCompleteFlag`
 Transmission complete flag, sets when transmission activity complete. bit 22

enumerator `kLPUART_RxDataRegFullFlag`
 Receive data register full flag, sets when the receive data buffer is full. bit 21

enumerator `kLPUART_IdleLineFlag`
 Idle line detect flag, sets when idle line detected. bit 20

enumerator `kLPUART_RxOverrunFlag`
 Receive Overrun, sets when new data is received before data is read from receive register. bit 19

enumerator `kLPUART_NoiseErrorFlag`
 Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets. bit 18

enumerator `kLPUART_FramingErrorFlag`
 Frame error flag, sets if logic 0 was detected where stop bit expected. bit 17

enumerator `kLPUART_ParityErrorFlag`
 If parity enabled, sets upon parity error detection. bit 16

enumerator `kLPUART_LinBreakFlag`
 LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled. bit 31

enumerator `kLPUART_RxActiveEdgeFlag`
 Receive pin active edge interrupt flag, sets when active edge detected. bit 30

enumerator `kLPUART_RxActiveFlag`
 Receiver Active Flag (RAF), sets at beginning of valid start. bit 24

enumerator `kLPUART_DataMatch1Flag`
 The next character to be read from LPUART_DATA matches MA1. bit 15

enumerator `kLPUART_DataMatch2Flag`
 The next character to be read from LPUART_DATA matches MA2. bit 14

enumerator `kLPUART_TxFifoEmptyFlag`
 TXEMPT bit, sets if transmit buffer is empty. bit 7

enumerator `kLPUART_RxFifoEmptyFlag`
 RXEMPT bit, sets if receive buffer is empty. bit 6

enumerator `kLPUART_TxFifoOverflowFlag`
 TXOF bit, sets if transmit buffer overflow occurred. bit 1

enumerator `kLPUART_RxFifoUnderflowFlag`
 RXUF bit, sets if receive buffer underflow occurred. bit 0

enumerator `kLPUART_AllClearFlags`

enumerator `kLPUART_AllFlags`

typedef enum *lpuart_parity_mode* lpuart_parity_mode_t
 LPUART parity mode.

typedef enum *lpuart_data_bits* lpuart_data_bits_t
 LPUART data bits count.

typedef enum *lpuart_stop_bit_count* lpuart_stop_bit_count_t
 LPUART stop bit count.

```
typedef enum _lpuart_transmit_cts_source lpuart_transmit_cts_source_t
    LPUART transmit CTS source.

typedef enum _lpuart_transmit_cts_config lpuart_transmit_cts_config_t
    LPUART transmit CTS configure.

typedef enum _lpuart_idle_type_select lpuart_idle_type_select_t
    LPUART idle flag type defines when the receiver starts counting.

typedef enum _lpuart_idle_config lpuart_idle_config_t
    LPUART idle detected configuration. This structure defines the number of idle characters
    that must be received before the IDLE flag is set.

typedef struct _lpuart_config lpuart_config_t
    LPUART configuration structure.

typedef struct _lpuart_transfer lpuart_transfer_t
    LPUART transfer structure.

typedef struct _lpuart_handle lpuart_handle_t

typedef void (*lpuart_transfer_callback_t)(LPUART_Type *base, lpuart_handle_t *handle,
status_t status, void *userData)
    LPUART transfer callback function.

typedef void (*lpuart_isr_t)(LPUART_Type *base, void *handle)

void *s_lpuartHandle[]

const IRQn_Type s_lpuartTxIRQ[]

lpuart_isr_t s_lpuartIsr[]

UART_RETRY_TIMES
    Retry times for waiting flag.

struct _lpuart_config
    #include <fsl_lpuart.h> LPUART configuration structure.
```

Public Members

```
uint32_t baudRate_Bps
    LPUART baud rate

lpuart_parity_mode_t parityMode
    Parity mode, disabled (default), even, odd

lpuart_data_bits_t dataBitsCount
    Data bits count, eight (default), seven

bool isMsb
    Data bits order, LSB (default), MSB

lpuart_stop_bit_count_t stopBitCount
    Number of stop bits, 1 stop bit (default) or 2 stop bits

uint8_t txFifoWatermark
    TX FIFO watermark

uint8_t rxFifoWatermark
    RX FIFO watermark
```

`bool enableRxRTS`
RX RTS enable

`bool enableTxCTS`
TX CTS enable

`lpuart_transmit_cts_source_t txCtsSource`
TX CTS source

`lpuart_transmit_cts_config_t txCtsConfig`
TX CTS configure

`lpuart_idle_type_select_t rxIdleType`
RX IDLE type.

`lpuart_idle_config_t rxIdleConfig`
RX IDLE configuration.

`bool enableTx`
Enable TX

`bool enableRx`
Enable RX

`struct __lpuart__transfer`
#include <fsl_lpuart.h> LPUART transfer structure.

Public Members

`size_t dataSize`
The byte count to be transfer.

`struct __lpuart__handle`
#include <fsl_lpuart.h> LPUART handle structure.

Public Members

`volatile size_t txDataSize`
Size of the remaining data to send.

`size_t txDataSizeAll`
Size of the data to send out.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t rxDataSizeAll`
Size of the data to receive.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

`volatile uint16_t rxRingBufferTail`
Index for the user to get data from the ring buffer.

`lpuart_transfer_callback_t callback`
Callback function.

`void *userData`
LPUART callback function parameter.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state.

`bool isSevenDataBits`
Seven data bits flag.

`bool is16bitData`
16bit data bits flag, only used for 9bit or 10bit data

`union __unnamed16__`

Public Members

`uint8_t *data`
The buffer of data to be transfer.

`uint8_t *rxData`
The buffer to receive data.

`uint16_t *rxData16`
The buffer to receive data.

`const uint8_t *txData`
The buffer of data to be sent.

`const uint16_t *txData16`
The buffer of data to be sent.

`union __unnamed18__`

Public Members

`const uint8_t *volatile txData`
Address of remaining data to send.

`const uint16_t *volatile txData16`
Address of remaining data to send.

`union __unnamed20__`

Public Members

`uint8_t *volatile rxData`
Address of remaining data to receive.

`uint16_t *volatile rxData16`
Address of remaining data to receive.

`union __unnamed22__`

Public Members

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`uint16_t *rxRingBuffer16`
Start address of the receiver ring buffer.

2.43 LPUART eDMA Driver

```
void LPUART__TransferCreateHandleEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                                     lpuart_edma_transfer_callback_t callback, void  
                                     *userData, edma_handle_t *txEdmaHandle,  
                                     edma_handle_t *rxEdmaHandle)
```

Initializes the LPUART handle which is used in transactional functions.

Note: This function disables all LPUART interrupts.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – Pointer to `lpuart_edma_handle_t` structure.
- `callback` – Callback function.
- `userData` – User data.
- `txEdmaHandle` – User requested DMA handle for TX DMA transfer.
- `rxEdmaHandle` – User requested DMA handle for RX DMA transfer.

```
status_t LPUART__SendEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                          lpuart_transfer_t *xfer)
```

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART eDMA transfer structure. See `lpuart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_LPUART_TxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
status_t LPUART__ReceiveEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle,  
                             lpuart_transfer_t *xfer)
```

Receives data using eDMA.

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.
- xfer – LPUART eDMA transfer structure, see `lpuart_transfer_t`.

Return values

- `kStatus__Success` – if succeed, others fail.
- `kStatus__LPUART_RxBusy` – Previous transfer ongoing.
- `kStatus__InvalidArgument` – Invalid argument.

`void LPUART__TransferAbortSendEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle)`
Aborts the sent data using eDMA.

This function aborts the sent data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.

`void LPUART__TransferAbortReceiveEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle)`
Aborts the received data using eDMA.

This function aborts the received data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to `lpuart_edma_handle_t` structure.

`status_t LPUART__TransferGetSendCountEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)`

Gets the number of bytes written to the LPUART TX register.

This function gets the number of bytes written to the LPUART TX register by DMA.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Send bytes count.

Return values

- `kStatus__NoTransferInProgress` – No send in progress.
- `kStatus__InvalidArgument` – Parameter is invalid.
- `kStatus__Success` – Get successfully through the parameter count;

`status_t LPUART__TransferGetReceiveCountEDMA(LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)`

Gets the number of received bytes.

This function gets the number of received bytes.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

`void LPUART_TransferEdmaHandleIRQ(LPUART_Type *base, void *lpuartEdmaHandle)`
LPUART eDMA IRQ handle function.

This function handles the LPUART tx complete IRQ request and invoke user callback. It is not set to static so that it can be used in user application.

Note: This function is used as default IRQ handler by double weak mechanism. If user's specific IRQ handler is implemented, make sure this function is invoked in the handler.

Parameters

- `base` – LPUART peripheral base address.
- `lpuartEdmaHandle` – LPUART handle pointer.

`FSL_LPUART_EDMA_DRIVER_VERSION`

LPUART EDMA driver version.

`typedef struct _lpuart_edma_handle lpuart_edma_handle_t`

`typedef void (*lpuart_edma_transfer_callback_t)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)`

LPUART transfer callback function.

`struct _lpuart_edma_handle`

`#include <fsl_lpuart_edma.h>` LPUART eDMA handle.

Public Members

`lpuart_edma_transfer_callback_t` callback

Callback function.

`void *userData`

LPUART callback function parameter.

`size_t rxDataSizeAll`

Size of the data to receive.

`size_t txDataSizeAll`

Size of the data to send out.

`edma_handle_t *txEdmaHandle`

The eDMA TX channel used.

`edma_handle_t *rxEdmaHandle`

The eDMA RX channel used.

`uint8_t nbytes`

eDMA minor byte transfer count initially configured.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.44 OCOTP: On Chip One-Time Programmable controller.

FSL_OCOTP_DRIVER_VERSION

OCOTP driver version.

_ocotp_status Error codes for the OCOTP driver.

Values:

enumerator kStatus_OCOTP_AccessError

eFuse and shadow register access error.

enumerator kStatus_OCOTP_CrcFail

CRC check failed.

enumerator kStatus_OCOTP_ReloadError

Error happens during reload shadow register.

enumerator kStatus_OCOTP_ProgramFail

Fuse programming failed.

enumerator kStatus_OCOTP_Locked

Fuse is locked and cannot be programmed.

typedef struct _ocotp_timing ocotp_timing_t

OCOTP timing structure. Note that, these value are used for calcalating the read/write timings. And the values should statisfy below rules:

Tsp_rd=(WAIT+1)/ipg_clk_freq should be $\geq 150\text{ns}$; Tsp_pgm=(RELAX+1)/ipg_clk_freq should be $\geq 100\text{ns}$; Trd = ((STROBE_READ+1)- 2*(RELAX_READ+1)) /ipg_clk_freq, The Trd is required to be larger than 40 ns. Tpgm = ((STROBE_PROG+1)- 2*(RELAX_PROG+1)) /ipg_clk_freq; The Tpgm should be configured within the range of $9000\text{ ns} < \text{Tpgm} < 11000\text{ ns}$;

void OCOTP_Init(OCOTP_Type *base, uint32_t srcClock_Hz)

Initializes OCOTP controller.

Parameters

- base – OCOTP peripheral base address.
- srcClock_Hz – source clock frequency in unit of Hz. When the macro FSL_FEATURE_OCOTP_HAS_TIMING_CTRL is defined as 0, this parameter is not used, application could pass in 0 in this case.

void OCOTP_Deinit(OCOTP_Type *base)

De-initializes OCOTP controller.

Return values

kStatus_Success – upon successful execution, error status otherwise.

static inline bool OCOTP_CheckBusyStatus(OCOTP_Type *base)

Checking the BUSY bit in CTRL register. Checking this BUSY bit will help confirm if the OCOTP controller is ready for access.

Parameters

- base – OCOTP peripheral base address.

Return values

true – for bit set and false for cleared.

static inline bool OCOTP_CheckErrorStatus(OCOTP_Type *base)

Checking the ERROR bit in CTRL register.

Parameters

- base – OCOTP peripheral base address.

Return values

true – for bit set and false for cleared.

static inline void OCOTP_ClearErrorStatus(OCOTP_Type *base)

Clear the error bit if this bit is set.

Parameters

- base – OCOTP peripheral base address.

status_t OCOTP_ReloadShadowRegister(OCOTP_Type *base)

Reload the shadow register. This function will help reload the shadow register without resetting the OCOTP module. Please make sure the OCOTP has been initialized before calling this API.

Parameters

- base – OCOTP peripheral base address.

Return values

- kStatus_Success – Reload success.
- kStatus_OCOTP_ReloadError – Reload failed.

uint32_t OCOTP_ReadFuseShadowRegister(OCOTP_Type *base, uint32_t address)

Read the fuse shadow register with the fuse address.

Deprecated:

Use OCOTP_ReadFuseShadowRegisterExt instead of this function.

Parameters

- base – OCOTP peripheral base address.
- address – the fuse address to be read from.

Returns

The read out data.

status_t OCOTP_ReadFuseShadowRegisterExt(OCOTP_Type *base, uint32_t address, uint32_t *data, uint8_t fuseWords)

Read the fuse shadow register from the fuse address.

This function reads fuse from address, how many words to read is specified by the parameter fuseWords. This function could read at most OCOTP_READ_FUSE_DATA_COUNT fuse word one time.

Parameters

- base – OCOTP peripheral base address.
- address – the fuse address to be read from.
- data – Data array to save the readout fuse value.
- fuseWords – How many words to read.

Return values

- kStatus_Success – Read success.

- `kStatus_Fail` – Error occurs during read.

`status_t` OCOTP_WriteFuseShadowRegister(OCOTP_Type *base, uint32_t address, uint32_t data)

Write the fuse shadow register with the fuse address and data. Please make sure the write address is not locked while calling this API.

Parameters

- `base` – OCOTP peripheral base address.
- `address` – the fuse address to be written.
- `data` – the value will be written to fuse address.

Return values

`write` – `status`, `kStatus_Success` for success and `kStatus_Fail` for failed.

`status_t` OCOTP_WriteFuseShadowRegisterWithLock(OCOTP_Type *base, uint32_t address, uint32_t data, bool lock)

Write the fuse shadow register and lock it.

Please make sure the write address is not locked while calling this API.

Some OCOTP controller supports ECC mode and redundancy mode (see reference manual for more details). OCOTP controller will auto select ECC or redundancy mode to program the fuse word according to fuse map definition. In ECC mode, the 32 fuse bits in one word can only be written once. In redundancy mode, the word can be written more than once as long as they are different fuse bits. Set parameter `lock` as true to force use ECC mode.

Parameters

- `base` – OCOTP peripheral base address.
- `address` – The fuse address to be written.
- `data` – The value will be written to fuse address.
- `lock` – Lock or unlock write fuse shadow register operation.

Return values

- `kStatus_Success` – Program and reload success.
- `kStatus_OCOTP_Locked` – The eFuse word is locked and cannot be programmed.
- `kStatus_OCOTP_ProgramFail` – eFuse word programming failed.
- `kStatus_OCOTP_ReloadError` – eFuse word programming success, but error happens during reload the values.
- `kStatus_OCOTP_AccessError` – Cannot access eFuse word.

static inline uint32_t OCOTP_GetVersion(OCOTP_Type *base)

Get the OCOTP controller version from the register.

Parameters

- `base` – OCOTP peripheral base address.

Return values

`return` – the version value.

OCOTP_READ_FUSE_DATA_COUNT

struct __ocotp_timing

`#include <fsl_ocotp.h>` OCOTP timing structure. Note that, these value are used for calculating the read/write timings. And the values should satisfy below rules:

Tsp_rd=(WAIT+1)/ipg_clk_freq should be $\geq 150\text{ns}$; Tsp_pgm=(RELAX+1)/ipg_clk_freq should be $\geq 100\text{ns}$; Trd = ((STROBE_READ+1)- 2*(RELAX_READ+1)) /ipg_clk_freq, The Trd is required to be larger than 40 ns. Tpgm = ((STROBE_PROG+1)- 2*(RELAX_PROG+1)) /ipg_clk_freq; The Tpgm should be configured within the range of $9000\text{ ns} < \text{Tpgm} < 11000\text{ ns}$;

Public Members

uint32_t wait

Wait time value to fill in the TIMING register.

uint32_t relax

Relax time value to fill in the TIMING register.

uint32_t strobe_prog

Storbe program time value to fill in the TIMING register.

uint32_t strobe_read

Storbe read time value to fill in the TIMING register.

2.45 OTFAD: On The Fly AES-128 Decryption Driver

void OTFAD_GetDefaultConfig(*otfad_config_t* *config)

OTFAD module initialization function.

Parameters

- config – OTFAD configuration.

void OTFAD_Init(OTFAD_Type *base, const *otfad_config_t* *config)

OTFAD module initialization function.

Parameters

- base – OTFAD base address.
- config – OTFAD configuration.

void OTFAD_Deinit(OTFAD_Type *base)

Deinitializes the OTFAD.

static inline uint32_t OTFAD_GetOperateMode(OTFAD_Type *base)

OTFAD module get operate mode.

Parameters

- base – OTFAD base address.

static inline uint32_t OTFAD_GetStatus(OTFAD_Type *base)

OTFAD module get status.

Parameters

- base – OTFAD base address.

status_t OTFAD_SetEncryptionConfig(OTFAD_Type *base, const *otfad_encryption_config_t* *config)

OTFAD module set encryption configuration.

Note: if enable keyblob process, the first 256 bytes external memory is use for keyblob data, so this region shouldn't be in OTFAD region.

Parameters

- base – OTFAD base address.
- config – encryption configuration.

status_t OTFAD_GetEncryptionConfig(OTFAD_Type *base, *otfad_encryption_config_t* *config)
OTFAD module get encryption configuration.

Note: if enable keyblob process, the first 256 bytes external memory is use for keyblob data, so this region shouldn't be in OTFAD region.

Parameters

- base – OTFAD base address.
- config – encryption configuration.

status_t OTFAD_HitDetermination(OTFAD_Type *base, uint32_t address, uint8_t *contextIndex)
OTFAD module do hit determination.

Parameters

- base – OTFAD base address.
- address – the physical address space assigned to the QuadSPI(FlexSPI) module.
- contextIndex – hitted context region index.

Returns

status, such as kStatus_Success or kStatus_OTFAD_ResRegAccessMode.

FSL_OTFAD_DRIVER_VERSION
Driver version.

Status codes for the OTFAD driver.

Values:

enumerator kStatus_OTFAD_ResRegAccessMode
Restricted register mode

enumerator kStatus_OTFAD_AddressError
End address less than start address

enumerator kStatus_OTFAD_RegionOverlap
the OTFAD does not support any form of memory region overlap, for system accesses that hit in multiple contexts or no contexts, the fetched data is simply bypassed

enumerator kStatus_OTFAD_RegionMiss
For accesses that hit in a single context, but not the selected one

OTFAD context type.

Values:

enumerator kOTFAD_Context_0
context 0

enumerator kOTFAD_Context_1
context 1

```
enumerator kOTFAD_Context_2
    context 2
enumerator kOTFAD_Context_3
    context 3
```

OTFAD operate mode.

Values:

```
enumerator kOTFAD_NRM
    Normal Mode
```

```
enumerator kOTFAD_SVM
    Security Violation Mode
```

```
enumerator kOTFAD_LDM
    Logically Disabled Mode
```

```
typedef struct _otfad_encryption_config otfad_encryption_config_t
    OTFAD encryption configuration structure.
```

```
typedef struct _otfad_config otfad_config_t
    OTFAD configuration structure.
```

```
struct _otfad_encryption_config
    #include <fsl_otfad.h> OTFAD encryption configuration structure.
```

Public Members

```
bool valid
    The context is valid or not

bool AESdecryption
    AES decryption enable

uint8_t readOnly
    read write attribute for the entire set of context registers

uint8_t contextIndex
    OTFAD context index

uint32_t startAddr
    Start address

uint32_t endAddr
    End address

uint32_t key[4]
    Encryption key

uint32_t counter[2]
    Encryption counter
```

```
struct _otfad_config
    #include <fsl_otfad.h> OTFAD configuration structure.
```

Public Members

`bool enableIntRequest`
Interrupt request enable

`bool forceError`
Forces the OTFAD's key blob error flag (SR[KBERR]) to be asserted

`bool forceSVM`
Force entry into SVM after a write

`bool forceLDM`
Force entry into LDM after a write

`bool keyBlobScramble`
Key blob KEK scrambling

`bool keyBlobProcess`
Key blob processing

`bool startKeyBlobProcessing`
key blob processing is initiated

`bool restrictedRegAccess`
Restricted register access enable

`bool enableOTFAD`
OTFAD has decryption enabled

2.46 PIT: Periodic Interrupt Timer

`void PIT_Init(PIT_Type *base, const pit_config_t *config)`

Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.

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

Parameters

- `base` – PIT peripheral base address
- `config` – Pointer to the user's PIT config structure

`void PIT_Deinit(PIT_Type *base)`

Gates the PIT clock and disables the PIT module.

Parameters

- `base` – PIT peripheral base address

`static inline void PIT_GetDefaultConfig(pit_config_t *config)`

Fills in the PIT configuration structure with the default settings.

The default values are as follows.

```
config->enableRunInDebug = false;
```

Parameters

- `config` – Pointer to the configuration structure.

static inline void PIT_SetTimerChainMode(PIT_Type *base, *pit_chnl_t* channel, bool enable)

Enables or disables chaining a timer with the previous timer.

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer n-1 has counted down to 0, counter n decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number which is chained with the previous timer
- enable – Enable or disable chain. true: Current timer is chained with the previous timer. false: Timer doesn't chain with other timers.

static inline void PIT_EnableInterrupts(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)

Enables the selected PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline void PIT_DisableInterrupts(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)

Disables the selected PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline uint32_t PIT_GetEnabledInterrupts(PIT_Type *base, *pit_chnl_t* channel)

Gets the enabled PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number

Returns

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

static inline uint32_t PIT_GetStatusFlags(PIT_Type *base, *pit_chnl_t* channel)

Gets the PIT status flags.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number

Returns

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

```
static inline void PIT_ClearStatusFlags(PIT_Type *base, pit_chnl_t channel, uint32_t mask)
```

Clears the PIT status flags.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- mask – The status flags to clear. This is a logical OR of members of the enumeration `pit_status_flags_t`

```
static inline void PIT_SetTimerPeriod(PIT_Type *base, pit_chnl_t channel, uint32_t count)
```

Sets the timer period in units of count.

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: Users can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- count – Timer period in units of ticks

```
static inline uint32_t PIT_GetCurrentTimerCount(PIT_Type *base, pit_chnl_t channel)
```

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: Users can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number

Returns

Current timer counting value in ticks

```
static inline void PIT_StartTimer(PIT_Type *base, pit_chnl_t channel)
```

Starts the timer counting.

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number.

```
static inline void PIT_StopTimer(PIT_Type *base, pit_chnl_t channel)
```

Stops the timer counting.

This function stops every timer counting. Timers reload their periods respectively after the next time they call the `PIT_DRV_StartTimer`.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number.

FSL_PIT_DRIVER_VERSION

PIT Driver Version 2.2.0.

enum __pit_chnl

List of PIT channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kPIT_Chnl_0

PIT channel number 0

enumerator kPIT_Chnl_1

PIT channel number 1

enumerator kPIT_Chnl_2

PIT channel number 2

enumerator kPIT_Chnl_3

PIT channel number 3

enum __pit_interrupt_enable

List of PIT interrupts.

Values:

enumerator kPIT_TimerInterruptEnable

Timer interrupt enable

enum __pit_status_flags

List of PIT status flags.

Values:

enumerator kPIT_TimerFlag

Timer flag

typedef enum __pit_chnl pit_chnl_t

List of PIT channels.

Note: Actual number of available channels is SoC dependent

typedef enum __pit_interrupt_enable pit_interrupt_enable_t

List of PIT interrupts.

typedef enum __pit_status_flags pit_status_flags_t

List of PIT status flags.

typedef struct __pit_config pit_config_t

PIT configuration structure.

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

The configuration structure can be made constant so it resides in flash.

uint64_t PIT_GetLifetimeTimerCount(PIT_Type *base)

Reads the current lifetime counter value.

The lifetime timer is a 64-bit timer which chains timer 0 and timer 1 together. Timer 0 and 1 are chained by calling the PIT_SetTimerChainMode before using this timer. The period of lifetime timer is equal to the “period of timer 0 * period of timer 1”. For the 64-bit value, the higher 32-bit has the value of timer 1, and the lower 32-bit has the value of timer 0.

Parameters

- base – PIT peripheral base address

Returns

Current lifetime timer value

struct __pit_config

#include <fsl_pit.h> PIT configuration structure.

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

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

2.47 PMU: Power Management Unit

uint32_t PMU_GetStatusFlags(PMU_Type *base)

Get PMU status flags.

Parameters

- base – PMU peripheral base address.

Returns

PMU status flags. It indicates if regulator output of 1P1, 3P0 and 2P5 is ok and brownout output of 1P1, 3P0 and 2P5 is detected.

static inline void PMU_1P1SetWeakReferenceSource(PMU_Type *base,
pmu_1p1_weak_reference_source_t option)

Selects the source for the reference voltage of the weak 1P1 regulator.

Parameters

- base – PMU peripheral base address.
- option – The option for reference voltage source, see to pmu_1p1_weak_reference_source_t.

static inline void PMU_1P1EnableWeakRegulator(PMU_Type *base, bool enable)

Enables the weak 1P1 regulator.

This regulator can be used when the main 1P1 regulator is disabled, under low-power conditions.

Parameters

- base – PMU peripheral base address.

- enable – Enable the feature or not.

static inline void PMU_1P1SetRegulatorOutputVoltage(PMU_Type *base, uint32_t value)

Adjust the 1P1 regulator output voltage.

Each LSB is worth 25mV. Programming examples are detailed below. Other output target voltages may be interpolated from these examples. Choices must be in this range:

- 0x1b(1.375V) >= output_trg >= 0x04(0.8V)
- 0x04 : 0.8V
- 0x10 : 1.1V (typical)
- 0x1b : 1.375V NOTE: There may be reduced chip functionality or reliability at the extremes of the programming range.

Parameters

- base – PMU peripheral base address.
- value – Setting value for the output.

static inline void PMU_1P1SetBrownoutOffsetVoltage(PMU_Type *base, uint32_t value)

Adjust the 1P1 regulator brownout offset voltage.

Control bits to adjust the regulator brownout offset voltage in 25mV steps. The reset brownout offset is 175mV below the programmed target code. Brownout target = OUTPUT_TRG - BO_OFFSET. Some steps may be irrelevant because of input supply limitations or load operation.

Parameters

- base – PMU peripheral base address.
- value – Setting value for the brownout offset. The available range is in 3-bit.

static inline void PMU_1P1EnablePullDown(PMU_Type *base, bool enable)

Enable the pull-down circuitry in the regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

static inline void PMU_1P1EnableCurrentLimit(PMU_Type *base, bool enable)

Enable the current-limit circuitry in the regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

static inline void PMU_1P1EnableBrownout(PMU_Type *base, bool enable)

Enable the brownout circuitry in the regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

static inline void PMU_1P1EnableOutput(PMU_Type *base, bool enable)

Enable the regulator output.

Parameters

- base – PMU peripheral base address.

- enable – Enable the feature or not.

static inline void PMU_3P0SetRegulatorOutputVoltage(PMU_Type *base, uint32_t value)

Adjust the 3P0 regulator output voltage.

Each LSB is worth 25mV. Programming examples are detailed below. Other output target voltages may be interpolated from these examples. Choices must be in this range:

- 0x00(2.625V) >= output_trg >= 0x1f(3.4V)
- 0x00 : 2.625V
- 0x0f : 3.0V (typical)
- 0x1f : 3.4V

Parameters

- base – PMU peripheral base address.
- value – Setting value for the output.

static inline void PMU_3P0SetVBusVoltageSource(PMU_Type *base,
 pmu_3p0_vbus_voltage_source_t option)

Select input voltage source for LDO_3P0.

Select input voltage source for LDO_3P0 from either USB_OTG1_VBUS or USB_OTG2_VBUS. If only one of the two VBUS voltages is present, it is automatically selected.

Parameters

- base – PMU peripheral base address.
- option – User-defined input voltage source for LDO_3P0.

static inline void PMU_3P0SetBrownoutOffsetVoltage(PMU_Type *base, uint32_t value)

Adjust the 3P0 regulator brownout offset voltage.

Control bits to adjust the 3P0 regulator brownout offset voltage in 25mV steps. The reset brown-offset is 175mV below the programmed target code. Brownout target = OUTPUT_TRG - BO_OFFSET. Some steps may be irrelevant because of input supply limitations or load operation.

Parameters

- base – PMU peripheral base address.
- value – Setting value for the brownout offset. The available range is in 3-bit.

static inline void PMU_3P0EnableCurrentLimit(PMU_Type *base, bool enable)

Enable the current-limit circuitry in the 3P0 regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

static inline void PMU_3P0EnableBrownout(PMU_Type *base, bool enable)

Enable the brownout circuitry in the 3P0 regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_3P0EnableOutput(PMU_Type *base, bool enable)
```

Enable the 3P0 regulator output.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_2P5EnableWeakRegulator(PMU_Type *base, bool enable)
```

Enables the weak 2P5 regulator.

This low power regulator is used when the main 2P5 regulator is disabled to keep the 2.5V output roughly at 2.5V. Scales directly with the value of VDDHIGH_IN.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_2P5SetRegulatorOutputVoltage(PMU_Type *base, uint32_t value)
```

Adjust the 1P1 regulator output voltage.

Each LSB is worth 25mV. Programming examples are detailed below. Other output target voltages may be interpolated from these examples. Choices must be in this range:

- 0x00(2.1V) >= output_trg >= 0x1f(2.875V)
- 0x00 : 2.1V
- 0x10 : 2.5V (typical)
- 0x1f : 2.875V NOTE: There may be reduced chip functionality or reliability at the extremes of the programming range.

Parameters

- base – PMU peripheral base address.
- value – Setting value for the output.

```
static inline void PMU_2P5SetBrownoutOffsetVoltage(PMU_Type *base, uint32_t value)
```

Adjust the 2P5 regulator brownout offset voltage.

Adjust the regulator brownout offset voltage in 25mV steps. The reset brown-offset is 175mV below the programmed target code. Brownout target = OUTPUT_TRG - BO_OFFSET. Some steps may be irrelevant because of input supply limitations or load operation.

Parameters

- base – PMU peripheral base address.
- value – Setting value for the brownout offset. The available range is in 3-bit.

```
static inline void PMU_2P5EnablePullDown(PMU_Type *base, bool enable)
```

Enable the pull-down circuitry in the 2P5 regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_2P1EnablePullDown(PMU_Type *base, bool enable)
```

Enable the pull-down circuitry in the 2P5 regulator.

Deprecated:

Do not use this function. It has been superceded by PMU_2P5EnablePullDown.

```
static inline void PMU_2P5EnableCurrentLimit(PMU_Type *base, bool enable)
```

Enable the current-limit circuitry in the 2P5 regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_2P5enableBrownout(PMU_Type *base, bool enable)
```

Enable the brownout circuitry in the 2P5 regulator.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_2P5EnableOutput(PMU_Type *base, bool enable)
```

Enable the 2P5 regulator output.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_CoreEnableIncreaseGateDrive(PMU_Type *base, bool enable)
```

Increase the gate drive on power gating FETs.

If set, increases the gate drive on power gating FETs to reduce leakage in the off state. Care must be taken to apply this bit only when the input supply voltage to the power FET is less than 1.1V. NOTE: This bit should only be used in low-power modes where the external input supply voltage is nominally 0.9V.

Parameters

- base – PMU peripheral base address.
- enable – Enable the feature or not.

```
static inline void PMU_CoreSetRegulatorVoltageRampRate(PMU_Type *base,  
                                                       pmu_core_reg_voltage_ramp_rate_t  
                                                       option)
```

Set the CORE regulator voltage ramp rate.

Parameters

- base – PMU peripheral base address.
- option – User-defined option for voltage ramp rate, see to pmu_core_reg_voltage_ramp_rate_t.

```
static inline void PMU_CoreSetSOCDomainVoltage(PMU_Type *base, uint32_t value)
```

Define the target voltage for the SOC power domain.

Define the target voltage for the SOC power domain. Single-bit increments reflect 25mV core voltage steps. Some steps may not be relevant because of input supply limitations or load operation.

- 0x00 : Power gated off.
- 0x01 : Target core voltage = 0.725V
- 0x02 : Target core voltage = 0.750V
- ...

- 0x10 : Target core voltage = 1.100V
- ...
- 0x1e : Target core voltage = 1.450V
- 0x1F : Power FET switched full on. No regulation. NOTE: This register is capable of programming an over-voltage condition on the device. Consult the datasheet Operating Ranges table for the allowed voltages.

Parameters

- base – PMU peripheral base address.
- value – Setting value for target voltage. 5-bit available

static inline void PMU_CoreSetARMCoreDomainVoltage(PMU_Type *base, uint32_t value)

Define the target voltage for the ARM Core power domain.

Define the target voltage for the ARM Core power domain. Single-bit increments reflect 25mV core voltage steps. Some steps may not be relevant because of input supply limitations or load operation.

- 0x00 : Power gated off.
- 0x01 : Target core voltage = 0.725V
- 0x02 : Target core voltage = 0.750V
- ...
- 0x10 : Target core voltage = 1.100V
- ...
- 0x1e : Target core voltage = 1.450V
- 0x1F : Power FET switched full on. No regulation. NOTE: This register is capable of programming an over-voltage condition on the device. Consult the datasheet Operating Ranges table for the allowed voltages.

Parameters

- base – PMU peripheral base address.
- value – Setting value for target voltage. 5-bit available

FSL_PMU_DRIVER_VERSION

PMU driver version.

Version 2.1.1.

PMU Status flags.

Values:

enumerator kPMU_1P1RegulatorOutputOK

Status bit that signals when the 1p1 regulator output is ok. 1 = regulator output > brownout target.

enumerator kPMU_1P1BrownoutOnOutput

Status bit that signals when a 1p1 brownout is detected on the regulator output.

enumerator kPMU_3P0RegulatorOutputOK

Status bit that signals when the 3p0 regulator output is ok. 1 = regulator output > brownout target.

enumerator kPMU_3P0BrownoutOnOutput

Status bit that signals when a 3p0 brownout is detected on the regulator output.

enumerator kPMU_2P5RegulatorOutputOK

Status bit that signals when the 2p5 regulator output is ok. 1 = regulator output > brownout target.

enumerator kPMU_2P5BrownoutOnOutput

Status bit that signals when a 2p5 brownout is detected on the regulator output.

enum __pmu_1p1_weak_reference_source

The source for the reference voltage of the weak 1P1 regulator.

Values:

enumerator kPMU_1P1WeakReferenceSourceAlt0

Weak-linreg output tracks low-power-bandgap voltage.

enumerator kPMU_1P1WeakReferenceSourceAlt1

Weak-linreg output tracks VDD_SOC_CAP voltage.

enum __pmu_3p0_vbus_voltage_source

Input voltage source for LDO_3P0 from USB VBus.

Values:

enumerator kPMU_3P0VBusVoltageSourceAlt0

USB_OTG1_VBUS - Utilize VBUS OTG1 for power.

enumerator kPMU_3P0VBusVoltageSourceAlt1

USB_OTG2_VBUS - Utilize VBUS OTG2 for power.

enum __pmu_core_reg_voltage_ramp_rate

Regulator voltage ramp rate.

Values:

enumerator kPMU_CoreRegVoltageRampRateFast

Fast.

enumerator kPMU_CoreRegVoltageRampRateMediumFast

Medium Fast.

enumerator kPMU_CoreRegVoltageRampRateMediumSlow

Medium Slow.

enumerator kPMU_CoreRegVoltageRampRateSlow

Slow.

enum __pmu_power_bandgap

Bandgap select.

Values:

enumerator kPMU_NormalPowerBandgap

Normal power bandgap.

enumerator kPMU_LowPowerBandgap

Low power bandgap.

typedef enum __pmu_1p1_weak_reference_source pmu_1p1_weak_reference_source_t

The source for the reference voltage of the weak 1P1 regulator.

```
typedef enum _pmu_3p0_vbus_voltage_source pmu_3p0_vbus_voltage_source_t
```

Input voltage source for LDO_3P0 from USB VBus.

```
typedef enum _pmu_core_reg_voltage_ramp_rate pmu_core_reg_voltage_ramp_rate_t
```

Regulator voltage ramp rate.

```
typedef enum _pmu_power_bandgap pmu_power_bandgap_t
```

Bandgap select.

2.48 PWM: Pulse Width Modulator

```
status_t PWM_Init(PWM_Type *base, pwm_submodule_t subModule, const pwm_config_t *config)
```

Ungates the PWM submodule clock and configures the peripheral for basic operation.

This API should be called at the beginning of the application using the PWM driver. When user select PWMX, user must choose edge aligned output, because there are some limitation on center aligned PWMX output. When output PWMX in center aligned mode, VAL1 register controls both PWM period and PWMX duty cycle, PWMA and PWMB output will be corrupted. But edge aligned PWMX output do not have such limit. In master reload counter initialization mode, PWM period is depended by period of set LDOK in submodule 0 because this operation will reload register. Submodule 0 counter initialization cannot be master sync or master reload.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- config – Pointer to user's PWM config structure.

Returns

kStatus_Success means success; else failed.

```
void PWM_Deinit(PWM_Type *base, pwm_submodule_t subModule)
```

Gate the PWM submodule clock.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to deinitialize

```
void PWM_GetDefaultConfig(pwm_config_t *config)
```

Fill in the PWM config struct with the default settings.

The default values are:

```
config->enableDebugMode = false;
config->enableWait = false;
config->reloadSelect = kPWM_LocalReload;
config->clockSource = kPWM_BusClock;
config->prescale = kPWM_Prescale_Divide_1;
config->initializationControl = kPWM_Initialize_LocalSync;
config->forceTrigger = kPWM_Force_Local;
config->reloadFrequency = kPWM_LoadEveryOpportunity;
config->reloadLogic = kPWM_ReloadImmediate;
config->pairOperation = kPWM_Independent;
```

Parameters

- config – Pointer to user's PWM config structure.

```
status_t PWM_SetupPwm(PWM_Type *base, pwm_submodule_t subModule, const  
    pwm_signal_param_t *chnlParams, uint8_t numOfChnls,  
    pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
```

Sets up the PWM signals for a PWM submodule.

The function initializes the submodule according to the parameters passed in by the user. The function also sets up the value compare registers to match the PWM signal requirements. If the dead time insertion logic is enabled, the pulse period is reduced by the dead time period specified by the user. When user select PWMX, user must choose edge aligned output, because there are some limitation on center aligned PWMX output. Due to edge aligned PWMX is negative true signal, need to configure PWMX active low true level to get correct duty cycle. The half cycle point will not be exactly in the middle of the PWM cycle when PWMX enabled.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- chnlParams – Array of PWM channel parameters to configure the channel(s).
- numOfChnls – Number of channels to configure, this should be the size of the array passed in. Array size should not be more than 3 as each submodule has 3 pins to output PWM.
- mode – PWM operation mode, options available in enumeration *pwm_mode_t*
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – PWM source clock of correspond submodule in Hz. If source clock of submodule1,2,3 is from submodule0 AUX_CLK, its source clock is submodule0 source clock divided with submodule0 prescaler value instead of submodule0 source clock.

Returns

Returns *kStatus_Fail* if there was error setting up the signal; *kStatus_Success* otherwise

```
status_t PWM_SetupPwmPhaseShift(PWM_Type *base, pwm_submodule_t subModule,  
    pwm_channels_t pwmChannel, uint32_t pwmFreq_Hz,  
    uint32_t srcClock_Hz, uint8_t shiftvalue, bool doSync)
```

Set PWM phase shift for PWM channel running on channel PWM_A, PWM_B which with 50% duty cycle.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – PWM channel to configure
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – PWM main counter clock in Hz.
- shiftvalue – Phase shift value, range in 0 ~ 50
- doSync – true: Set LDOK bit for the submodule list; false: LDOK bit don't set, need to call PWM_SetPwmLdok to sync update.

Returns

Returns *kStatus_Fail* if there was error setting up the signal; *kStatus_Success* otherwise

```
void PWM_UpdatePwmDutycycle(PWM_Type *base, pwm_submodule_t subModule,  
                           pwm_channels_t pwmSignal, pwm_mode_t currPwmMode,  
                           uint8_t dutyCyclePercent)
```

Updates the PWM signal's dutycycle.

The function updates the PWM dutycycle to the new value that is passed in. If the dead time insertion logic is enabled then the pulse period is reduced by the dead time period specified by the user.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmSignal – Signal (PWM A, PWM B, PWM X) to update
- currPwmMode – The current PWM mode set during PWM setup
- dutyCyclePercent – New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

```
void PWM_UpdatePwmDutycycleHighAccuracy(PWM_Type *base, pwm_submodule_t subModule,  
                                         pwm_channels_t pwmSignal, pwm_mode_t  
                                         currPwmMode, uint16_t dutyCycle)
```

Updates the PWM signal's dutycycle with 16-bit accuracy.

The function updates the PWM dutycycle to the new value that is passed in. If the dead time insertion logic is enabled then the pulse period is reduced by the dead time period specified by the user.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmSignal – Signal (PWM A, PWM B, PWM X) to update
- currPwmMode – The current PWM mode set during PWM setup
- dutyCycle – New PWM pulse width, value should be between 0 to 65535 0=inactive signal(0% duty cycle)... 65535=active signal (100% duty cycle)

```
void PWM_UpdatePwmPeriodAndDutycycle(PWM_Type *base, pwm_submodule_t subModule,  
                                      pwm_channels_t pwmSignal, pwm_mode_t  
                                      currPwmMode, uint16_t pulseCnt, uint16_t  
                                      dutyCycle)
```

Update the PWM signal's period and dutycycle for a PWM submodule.

The function updates PWM signal period generated by a specific submodule according to the parameters passed in by the user. This function can also set dutycycle whether you want to keep original dutycycle or update new dutycycle. Call this function in local sync control mode because PWM period is depended by

INIT and VAL1 register of each submodule. In master sync initialization control mode, call this function to update INIT and VAL1 register of all submodule because PWM period is depended by INIT and VAL1 register in submodule0. If the dead time insertion logic is enabled, the pulse period is reduced by the dead time period specified by the user. PWM signal will not be generated if its period is less than dead time duration.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure

- `pwmSignal` – Signal (PWM A or PWM B) to update
- `currPwmMode` – The current PWM mode set during PWM setup, options available in enumeration `pwm_mode_t`
- `pulseCnt` – New PWM period, value should be between 0 to 65535 0=minimum PWM period... 65535=maximum PWM period
- `dutyCycle` – New PWM pulse width of channel, value should be between 0 to 65535 0=inactive signal(0% duty cycle)... 65535=active signal (100% duty cycle) You can keep original duty cycle or update new duty cycle

```
static inline void PWM_EnableInterrupts(PWM_Type *base, pwm_submodule_t subModule,  
                                       uint32_t mask)
```

Enables the selected PWM interrupts.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `pwm_interrupt_enable_t`

```
static inline void PWM_DisableInterrupts(PWM_Type *base, pwm_submodule_t subModule,  
                                       uint32_t mask)
```

Disables the selected PWM interrupts.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `pwm_interrupt_enable_t`

```
static inline uint32_t PWM_GetEnabledInterrupts(PWM_Type *base, pwm_submodule_t  
                                              subModule)
```

Gets the enabled PWM interrupts.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure

Returns

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

```
static inline void PWM_DMAFIFOWatermarkControl(PWM_Type *base, pwm_submodule_t  
                                              subModule, pwm_watermark_control_t  
                                              pwm_watermark_control)
```

Capture DMA Enable Source Select.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `pwm_watermark_control` – PWM FIFO watermark and control

```
static inline void PWM_DMACaptureSourceSelect(PWM_Type *base, pwm_submodule_t  
                                              subModule, pwm_dma_source_select_t  
                                              pwm_dma_source_select)
```

Capture DMA Enable Source Select.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwm_dma_source_select – PWM capture DMA enable source select

```
static inline void PWM_EnableDMACapture(PWM_Type *base, pwm_submodule_t subModule,  
                                         uint16_t mask, bool activate)
```

Enables or disables the selected PWM DMA Capture read request.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- mask – The DMA to enable or disable. This is a logical OR of members of the enumeration pwm_dma_enable_t
- activate – true: Enable DMA read request; false: Disable DMA read request

```
static inline void PWM_EnableDMAWrite(PWM_Type *base, pwm_submodule_t subModule, bool  
                                       activate)
```

Enables or disables the PWM DMA write request.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- activate – true: Enable DMA write request; false: Disable DMA write request

```
static inline uint32_t PWM_GetStatusFlags(PWM_Type *base, pwm_submodule_t subModule)
```

Gets the PWM status flags.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure

Returns

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

```
static inline void PWM_ClearStatusFlags(PWM_Type *base, pwm_submodule_t subModule,  
                                         uint32_t mask)
```

Clears the PWM status flags.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- mask – The status flags to clear. This is a logical OR of members of the enumeration pwm_status_flags_t

```
static inline void PWM_StartTimer(PWM_Type *base, uint8_t subModulesToStart)
```

Starts the PWM counter for a single or multiple submodules.

Sets the Run bit which enables the clocks to the PWM submodule. This function can start multiple submodules at the same time.

Parameters

- base – PWM peripheral base address

- subModulesToStart – PWM submodules to start. This is a logical OR of members of the enumeration pwm_module_control_t

static inline void PWM_StopTimer(PWM_Type *base, uint8_t subModulesToStop)

Stops the PWM counter for a single or multiple submodules.

Clears the Run bit which resets the submodule's counter. This function can stop multiple submodules at the same time.

Parameters

- base – PWM peripheral base address
- subModulesToStop – PWM submodules to stop. This is a logical OR of members of the enumeration pwm_module_control_t

FSL_PWM_DRIVER_VERSION

Version 2.9.1

enum __pwm_submodule

List of PWM submodules.

Values:

enumerator kPWM_Module_0

Submodule 0

enumerator kPWM_Module_1

Submodule 1

enumerator kPWM_Module_2

Submodule 2

enum __pwm_channels

List of PWM channels in each module.

Values:

enumerator kPWM_PwmB

enumerator kPWM_PwmA

enumerator kPWM_PwmX

enum __pwm_value_register

List of PWM value registers.

Values:

enumerator kPWM_ValueRegister_0

PWM Value0 register

enumerator kPWM_ValueRegister_1

PWM Value1 register

enumerator kPWM_ValueRegister_2

PWM Value2 register

enumerator kPWM_ValueRegister_3

PWM Value3 register

enumerator kPWM_ValueRegister_4

PWM Value4 register

enumerator kPWM_ValueRegister_5

PWM Value5 register

enum _pwm_value_register_mask

List of PWM value registers mask.

Values:

enumerator kPWM_ValueRegisterMask_0

PWM Value0 register mask

enumerator kPWM_ValueRegisterMask_1

PWM Value1 register mask

enumerator kPWM_ValueRegisterMask_2

PWM Value2 register mask

enumerator kPWM_ValueRegisterMask_3

PWM Value3 register mask

enumerator kPWM_ValueRegisterMask_4

PWM Value4 register mask

enumerator kPWM_ValueRegisterMask_5

PWM Value5 register mask

enum _pwm_clock_source

PWM clock source selection.

Values:

enumerator kPWM_BusClock

Device specific IPBus clock, refer reference manual for frequency

enumerator kPWM_ExternalClock

EXT_CLK is used as the clock

enumerator kPWM_Submodule0Clock

Clock of the submodule 0 (AUX_CLK) is used as the source clock

enum _pwm_clock_prescale

PWM prescaler factor selection for clock source.

Values:

enumerator kPWM_Prescale_Divide_1

PWM clock frequency = $f_{clk}/1$

enumerator kPWM_Prescale_Divide_2

PWM clock frequency = $f_{clk}/2$

enumerator kPWM_Prescale_Divide_4

PWM clock frequency = $f_{clk}/4$

enumerator kPWM_Prescale_Divide_8

PWM clock frequency = $f_{clk}/8$

enumerator kPWM_Prescale_Divide_16

PWM clock frequency = $f_{clk}/16$

enumerator kPWM_Prescale_Divide_32

PWM clock frequency = $f_{clk}/32$

enumerator kPWM_Prescale_Divide_64

PWM clock frequency = $f_{clk}/64$

enumerator kPWM_Prescale_Divide_128

PWM clock frequency = fclk/128

enum __pwm_force_output_trigger

Options that can trigger a PWM FORCE_OUT.

Values:

enumerator kPWM_Force_Local

The local force signal, CTRL2[FORCE], from the submodule is used to force updates

enumerator kPWM_Force_Master

The master force signal from submodule 0 is used to force updates

enumerator kPWM_Force_LocalReload

The local reload signal from this submodule is used to force updates without regard to the state of LDOK

enumerator kPWM_Force_MasterReload

The master reload signal from submodule 0 is used to force updates if LDOK is set

enumerator kPWM_Force_LocalSync

The local sync signal from this submodule is used to force updates

enumerator kPWM_Force_MasterSync

The master sync signal from submodule0 is used to force updates

enumerator kPWM_Force_External

The external force signal, EXT_FORCE, from outside the PWM module causes updates

enumerator kPWM_Force_ExternalSync

The external sync signal, EXT_SYNC, from outside the PWM module causes updates

enum __pwm_output_state

PWM channel output status.

Values:

enumerator kPWM_HighState

The output state of PWM channel is high

enumerator kPWM_LowState

The output state of PWM channel is low

enumerator kPWM_NormalState

The output state of PWM channel is normal

enumerator kPWM_InvertState

The output state of PWM channel is invert

enumerator kPWM_MaskState

The output state of PWM channel is mask

enum __pwm_init_source

PWM counter initialization options.

Values:

enumerator kPWM_Initialize_LocalSync

Local sync causes initialization

enumerator kPWM_Initialize_MasterReload

Master reload from submodule 0 causes initialization

enumerator kPWM_Initialize_MasterSync

Master sync from submodule 0 causes initialization

enumerator kPWM_Initialize_ExtSync

EXT_SYNC causes initialization

enum __pwm_load_frequency

PWM load frequency selection.

Values:

enumerator kPWM_LoadEveryOpportunity

Every PWM opportunity

enumerator kPWM_LoadEvery2Opportunity

Every 2 PWM opportunities

enumerator kPWM_LoadEvery3Opportunity

Every 3 PWM opportunities

enumerator kPWM_LoadEvery4Opportunity

Every 4 PWM opportunities

enumerator kPWM_LoadEvery5Opportunity

Every 5 PWM opportunities

enumerator kPWM_LoadEvery6Opportunity

Every 6 PWM opportunities

enumerator kPWM_LoadEvery7Opportunity

Every 7 PWM opportunities

enumerator kPWM_LoadEvery8Opportunity

Every 8 PWM opportunities

enumerator kPWM_LoadEvery9Opportunity

Every 9 PWM opportunities

enumerator kPWM_LoadEvery10Opportunity

Every 10 PWM opportunities

enumerator kPWM_LoadEvery11Opportunity

Every 11 PWM opportunities

enumerator kPWM_LoadEvery12Opportunity

Every 12 PWM opportunities

enumerator kPWM_LoadEvery13Opportunity

Every 13 PWM opportunities

enumerator kPWM_LoadEvery14Opportunity

Every 14 PWM opportunities

enumerator kPWM_LoadEvery15Opportunity

Every 15 PWM opportunities

enumerator kPWM_LoadEvery16Opportunity

Every 16 PWM opportunities

enum __pwm_fault_input

List of PWM fault selections.

Values:

enumerator kPWM_Fault_0

Fault 0 input pin

enumerator kPWM_Fault_1

Fault 1 input pin

enumerator kPWM_Fault_2

Fault 2 input pin

enumerator kPWM_Fault_3

Fault 3 input pin

enum __pwm_fault_disable

List of PWM fault disable mapping selections.

Values:

enumerator kPWM_FaultDisable_0

Fault 0 disable mapping

enumerator kPWM_FaultDisable_1

Fault 1 disable mapping

enumerator kPWM_FaultDisable_2

Fault 2 disable mapping

enumerator kPWM_FaultDisable_3

Fault 3 disable mapping

enum __pwm_fault_channels

List of PWM fault channels.

Values:

enumerator kPWM_faultchannel_0

enum __pwm_input_capture_edge

PWM capture edge select.

Values:

enumerator kPWM_Disable

Disabled

enumerator kPWM_FallingEdge

Capture on falling edge only

enumerator kPWM_RisingEdge

Capture on rising edge only

enumerator kPWM_RiseAndFallEdge

Capture on rising or falling edge

enum __pwm_force_signal

PWM output options when a FORCE_OUT signal is asserted.

Values:

enumerator kPWM_UsePwm

Generated PWM signal is used by the deadtime logic.

enumerator kPWM_InvertedPwm

Inverted PWM signal is used by the deadtime logic.

enumerator kPWM_SoftwareControl

Software controlled value is used by the deadtime logic.

enumerator kPWM_UseExternal

PWM_EXT_A signal is used by the deadtime logic.

enum __pwm_chnl_pair_operation

Options available for the PWM A & B pair operation.

Values:

enumerator kPWM_Independent

PWM A & PWM B operate as 2 independent channels

enumerator kPWM_ComplementaryPwmA

PWM A & PWM B are complementary channels, PWM A generates the signal

enumerator kPWM_ComplementaryPwmB

PWM A & PWM B are complementary channels, PWM B generates the signal

enum __pwm_register_reload

Options available on how to load the buffered-registers with new values.

Values:

enumerator kPWM_ReloadImmediate

Buffered-registers get loaded with new values as soon as LDOK bit is set

enumerator kPWM_ReloadPwmHalfCycle

Registers loaded on a PWM half cycle

enumerator kPWM_ReloadPwmFullCycle

Registers loaded on a PWM full cycle

enumerator kPWM_ReloadPwmHalfAndFullCycle

Registers loaded on a PWM half & full cycle

enum __pwm_fault_recovery_mode

Options available on how to re-enable the PWM output when recovering from a fault.

Values:

enumerator kPWM_NoRecovery

PWM output will stay inactive

enumerator kPWM_RecoverHalfCycle

PWM output re-enabled at the first half cycle

enumerator kPWM_RecoverFullCycle

PWM output re-enabled at the first full cycle

enumerator kPWM_RecoverHalfAndFullCycle

PWM output re-enabled at the first half or full cycle

enum __pwm_interrupt_enable

List of PWM interrupt options.

Values:

enumerator kPWM_CompareVal0InterruptEnable

PWM VAL0 compare interrupt

enumerator kPWM_CompareVal1InterruptEnable

PWM VAL1 compare interrupt

enumerator kPWM__CompareVal2InterruptEnable
PWM VAL2 compare interrupt

enumerator kPWM__CompareVal3InterruptEnable
PWM VAL3 compare interrupt

enumerator kPWM__CompareVal4InterruptEnable
PWM VAL4 compare interrupt

enumerator kPWM__CompareVal5InterruptEnable
PWM VAL5 compare interrupt

enumerator kPWM__CaptureX0InterruptEnable
PWM capture X0 interrupt

enumerator kPWM__CaptureX1InterruptEnable
PWM capture X1 interrupt

enumerator kPWM__CaptureB0InterruptEnable
PWM capture B0 interrupt

enumerator kPWM__CaptureB1InterruptEnable
PWM capture B1 interrupt

enumerator kPWM__CaptureA0InterruptEnable
PWM capture A0 interrupt

enumerator kPWM__CaptureA1InterruptEnable
PWM capture A1 interrupt

enumerator kPWM__ReloadInterruptEnable
PWM reload interrupt

enumerator kPWM__ReloadErrorInterruptEnable
PWM reload error interrupt

enumerator kPWM__Fault0InterruptEnable
PWM fault 0 interrupt

enumerator kPWM__Fault1InterruptEnable
PWM fault 1 interrupt

enumerator kPWM__Fault2InterruptEnable
PWM fault 2 interrupt

enumerator kPWM__Fault3InterruptEnable
PWM fault 3 interrupt

enum __pwm_status_flags
List of PWM status flags.

Values:

enumerator kPWM__CompareVal0Flag
PWM VAL0 compare flag

enumerator kPWM__CompareVal1Flag
PWM VAL1 compare flag

enumerator kPWM__CompareVal2Flag
PWM VAL2 compare flag

enumerator kPWM__CompareVal3Flag

PWM VAL3 compare flag

enumerator kPWM__CompareVal4Flag

PWM VAL4 compare flag

enumerator kPWM__CompareVal5Flag

PWM VAL5 compare flag

enumerator kPWM__CaptureX0Flag

PWM capture X0 flag

enumerator kPWM__CaptureX1Flag

PWM capture X1 flag

enumerator kPWM__CaptureB0Flag

PWM capture B0 flag

enumerator kPWM__CaptureB1Flag

PWM capture B1 flag

enumerator kPWM__CaptureA0Flag

PWM capture A0 flag

enumerator kPWM__CaptureA1Flag

PWM capture A1 flag

enumerator kPWM__ReloadFlag

PWM reload flag

enumerator kPWM__ReloadErrorFlag

PWM reload error flag

enumerator kPWM__RegUpdatedFlag

PWM registers updated flag

enumerator kPWM__Fault0Flag

PWM fault 0 flag

enumerator kPWM__Fault1Flag

PWM fault 1 flag

enumerator kPWM__Fault2Flag

PWM fault 2 flag

enumerator kPWM__Fault3Flag

PWM fault 3 flag

enum __pwm_dma_enable

List of PWM DMA options.

Values:

enumerator kPWM__CaptureX0DMAEnable

PWM capture X0 DMA

enumerator kPWM__CaptureX1DMAEnable

PWM capture X1 DMA

enumerator kPWM__CaptureB0DMAEnable

PWM capture B0 DMA

enumerator kPWM_CaptureB1DMAEnable

PWM capture B1 DMA

enumerator kPWM_CaptureA0DMAEnable

PWM capture A0 DMA

enumerator kPWM_CaptureA1DMAEnable

PWM capture A1 DMA

enum __pwm_dma_source_select

List of PWM capture DMA enable source select.

Values:

enumerator kPWM_DMAResourceDisable

Read DMA requests disabled

enumerator kPWM_DMAWatermarksEnable

Exceeding a FIFO watermark sets the DMA read request

enumerator kPWM_DMALocalSync

A local sync (VAL1 matches counter) sets the read DMA request

enumerator kPWM_DMALocalReload

A local reload (STS[RF] being set) sets the read DMA request

enum __pwm_watermark_control

PWM FIFO Watermark AND Control.

Values:

enumerator kPWM_FIFOWatermarksOR

Selected FIFO watermarks are OR'ed together

enumerator kPWM_FIFOWatermarksAND

Selected FIFO watermarks are AND'ed together

enum __pwm_mode

PWM operation mode.

Values:

enumerator kPWM_SignedCenterAligned

Signed center-aligned

enumerator kPWM_CenterAligned

Unsigned center-aligned

enumerator kPWM_SignedEdgeAligned

Signed edge-aligned

enumerator kPWM_EdgeAligned

Unsigned edge-aligned

enum __pwm_level_select

PWM output pulse mode, high-true or low-true.

Values:

enumerator kPWM_HighTrue

High level represents “on” or “active” state

enumerator kPWM_LowTrue

Low level represents “on” or “active” state

enum `_pwm_fault_state`

PWM output fault status.

Values:

enumerator `kPWM_PwmFaultState0`

Output is forced to logic 0 state prior to consideration of output polarity control.

enumerator `kPWM_PwmFaultState1`

Output is forced to logic 1 state prior to consideration of output polarity control.

enumerator `kPWM_PwmFaultState2`

Output is tristated.

enumerator `kPWM_PwmFaultState3`

Output is tristated.

enum `_pwm_reload_source_select`

PWM reload source select.

Values:

enumerator `kPWM_LocalReload`

The local reload signal is used to reload registers

enumerator `kPWM_MasterReload`

The master reload signal (from submodule 0) is used to reload

enum `_pwm_fault_clear`

PWM fault clearing options.

Values:

enumerator `kPWM_Automatic`

Automatic fault clearing

enumerator `kPWM_ManualNormal`

Manual fault clearing with no fault safety mode

enumerator `kPWM_ManualSafety`

Manual fault clearing with fault safety mode

enum `_pwm_module_control`

Options for submodule master control operation.

Values:

enumerator `kPWM_Control_Module_0`

Control submodule 0's start/stop,buffer reload operation

enumerator `kPWM_Control_Module_1`

Control submodule 1's start/stop,buffer reload operation

enumerator `kPWM_Control_Module_2`

Control submodule 2's start/stop,buffer reload operation

enumerator `kPWM_Control_Module_3`

Control submodule 3's start/stop,buffer reload operation

typedef enum `_pwm_submodule` `pwm_submodule_t`

List of PWM submodules.

typedef enum `_pwm_channels` `pwm_channels_t`

List of PWM channels in each module.

`typedef enum _pwm_value_register pwm_value_register_t`

List of PWM value registers.

`typedef enum _pwm_clock_source pwm_clock_source_t`

PWM clock source selection.

`typedef enum _pwm_clock_prescale pwm_clock_prescale_t`

PWM prescaler factor selection for clock source.

`typedef enum _pwm_force_output_trigger pwm_force_output_trigger_t`

Options that can trigger a PWM FORCE_OUT.

`typedef enum _pwm_output_state pwm_output_state_t`

PWM channel output status.

`typedef enum _pwm_init_source pwm_init_source_t`

PWM counter initialization options.

`typedef enum _pwm_load_frequency pwm_load_frequency_t`

PWM load frequency selection.

`typedef enum _pwm_fault_input pwm_fault_input_t`

List of PWM fault selections.

`typedef enum _pwm_fault_disable pwm_fault_disable_t`

List of PWM fault disable mapping selections.

`typedef enum _pwm_fault_channels pwm_fault_channels_t`

List of PWM fault channels.

`typedef enum _pwm_input_capture_edge pwm_input_capture_edge_t`

PWM capture edge select.

`typedef enum _pwm_force_signal pwm_force_signal_t`

PWM output options when a FORCE_OUT signal is asserted.

`typedef enum _pwm_chnl_pair_operation pwm_chnl_pair_operation_t`

Options available for the PWM A & B pair operation.

`typedef enum _pwm_register_reload pwm_register_reload_t`

Options available on how to load the buffered-registers with new values.

`typedef enum _pwm_fault_recovery_mode pwm_fault_recovery_mode_t`

Options available on how to re-enable the PWM output when recovering from a fault.

`typedef enum _pwm_interrupt_enable pwm_interrupt_enable_t`

List of PWM interrupt options.

`typedef enum _pwm_status_flags pwm_status_flags_t`

List of PWM status flags.

`typedef enum _pwm_dma_enable pwm_dma_enable_t`

List of PWM DMA options.

`typedef enum _pwm_dma_source_select pwm_dma_source_select_t`

List of PWM capture DMA enable source select.

`typedef enum _pwm_watermark_control pwm_watermark_control_t`

PWM FIFO Watermark AND Control.

`typedef enum _pwm_mode pwm_mode_t`

PWM operation mode.

```
typedef enum _pwm_level_select pwm_level_select_t
```

PWM output pulse mode, high-true or low-true.

```
typedef enum _pwm_fault_state pwm_fault_state_t
```

PWM output fault status.

```
typedef enum _pwm_reload_source_select pwm_reload_source_select_t
```

PWM reload source select.

```
typedef enum _pwm_fault_clear pwm_fault_clear_t
```

PWM fault clearing options.

```
typedef enum _pwm_module_control pwm_module_control_t
```

Options for submodule master control operation.

```
typedef struct _pwm_signal_param pwm_signal_param_t
```

Structure for the user to define the PWM signal characteristics.

```
typedef struct _pwm_config pwm_config_t
```

PWM config structure.

This structure holds the configuration settings for the PWM peripheral. To initialize this structure to reasonable defaults, call the `PWM_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

```
typedef struct _pwm_fault_input_filter_param pwm_fault_input_filter_param_t
```

Structure for the user to configure the fault input filter.

```
typedef struct _pwm_fault_param pwm_fault_param_t
```

Structure is used to hold the parameters to configure a PWM fault.

```
typedef struct _pwm_input_capture_param pwm_input_capture_param_t
```

Structure is used to hold parameters to configure the capture capability of a signal pin.

```
void PWM_SetupInputCapture(PWM_Type *base, pwm_submodule_t subModule,  
                           pwm_channels_t pwmChannel, const  
                           pwm_input_capture_param_t *inputCaptureParams)
```

Sets up the PWM input capture.

Each PWM submodule has 3 pins that can be configured for use as input capture pins. This function sets up the capture parameters for each pin and enables the pin for input capture operation.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `pwmChannel` – Channel in the submodule to setup
- `inputCaptureParams` – Parameters passed in to set up the input pin

```
void PWM_SetupFaultInputFilter(PWM_Type *base, const pwm_fault_input_filter_param_t  
                              *faultInputFilterParams)
```

Sets up the PWM fault channel 0 input filter.

Parameters

- `base` – PWM peripheral base address
- `faultInputFilterParams` – Parameters passed in to set up the fault input filter.

```
void PWM_SetupFaultInputFilterExt(PWM_Type *base, pwm_fault_channels_t faultChannel,  
                                const pwm_fault_input_filter_param_t  
                                *faultInputFilterParams)
```

Sets up the PWM fault input filter.

Parameters

- base – PWM peripheral base address
- faultChannel – PWM fault channel to configure.
- faultInputFilterParams – Parameters passed in to set up the fault input filter.

```
void PWM_SetupFaults(PWM_Type *base, pwm_fault_input_t faultNum, const  
                    pwm_fault_param_t *faultParams)
```

Sets up the PWM fault channel 0 protection.

Parameters

- base – PWM peripheral base address
- faultNum – PWM fault to configure.
- faultParams – Pointer to the PWM fault config structure

```
void PWM_SetupFaultsExt(PWM_Type *base, pwm_fault_channels_t faultChannel,  
                       pwm_fault_input_t faultNum, const pwm_fault_param_t  
                       *faultParams)
```

Sets up the PWM fault protection.

Parameters

- base – PWM peripheral base address
- faultChannel – PWM fault channel to configure.
- faultNum – PWM fault to configure.
- faultParams – Pointer to the PWM fault config structure

```
void PWM_FaultDefaultConfig(pwm_fault_param_t *config)
```

Fill in the PWM fault config struct with the default settings.

The default values are:

```
config->faultClearingMode = kPWM_Automatic;  
config->faultLevel = false;  
config->enableCombinationalPath = true;  
config->recoverMode = kPWM_NoRecovery;
```

Parameters

- config – Pointer to user's PWM fault config structure.

```
void PWM_SetupForceSignal(PWM_Type *base, pwm_submodule_t subModule, pwm_channels_t  
                         pwmChannel, pwm_force_signal_t mode)
```

Selects the signal to output on a PWM pin when a FORCE_OUT signal is asserted.

The user specifies which channel to configure by supplying the submodule number and whether to modify PWM A or PWM B within that submodule.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – Channel to configure

- mode – Signal to output when a FORCE_OUT is triggered

```
static inline void PWM_SetVALxValue(PWM_Type *base, pwm_submodule_t subModule,  
                                   pwm_value_register_t valueRegister, uint16_t value)
```

Set the PWM VALx registers.

This function allows the user to write value into VAL registers directly. And it will destroying the PWM clock period set by the PWM_SetupPwm()/PWM_SetupPwmPhaseShift() functions. Due to VALx registers are buffered, the new value will not active unless call PWM_SetPwmLdok() and the reload point is reached.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- valueRegister – VALx register that will be written new value
- value – Value that will be written into VALx register

```
static inline uint16_t PWM_GetVALxValue(PWM_Type *base, pwm_submodule_t subModule,  
                                       pwm_value_register_t valueRegister)
```

Get the PWM VALx registers.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- valueRegister – VALx register that will be read value

Returns

The VALx register value

```
static inline void PWM_OutputTriggerEnable(PWM_Type *base, pwm_submodule_t subModule,  
                                           pwm_value_register_t valueRegister, bool activate)
```

Enables or disables the PWM output trigger.

This function allows the user to enable or disable the PWM trigger. The PWM has 2 triggers. Trigger 0 is activated when the counter matches VAL 0, VAL 2, or VAL 4 register. Trigger 1 is activated when the counter matches VAL 1, VAL 3, or VAL 5 register.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- valueRegister – Value register that will activate the trigger
- activate – true: Enable the trigger; false: Disable the trigger

```
static inline void PWM_ActivateOutputTrigger(PWM_Type *base, pwm_submodule_t subModule,  
                                             uint16_t valueRegisterMask)
```

Enables the PWM output trigger.

This function allows the user to enable one or more (VAL0-5) PWM trigger.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- valueRegisterMask – Value register mask that will activate one or more (VAL0-5) trigger enumeration `_pwm_value_register_mask`

```
static inline void PWM_DeactivateOutputTrigger(PWM_Type *base, pwm_submodule_t
                                              submodule, uint16_t valueRegisterMask)
```

Disables the PWM output trigger.

This function allows the user to disables one or more (VAL0-5) PWM trigger.

Parameters

- base – PWM peripheral base address
- submodule – PWM submodule to configure
- valueRegisterMask – Value register mask that will Deactivate one or more (VAL0-5) trigger enumeration `_pwm_value_register_mask`

```
static inline void PWM_SetupSwCtrlOut(PWM_Type *base, pwm_submodule_t submodule,
                                      pwm_channels_t pwmChannel, bool value)
```

Sets the software control output for a pin to high or low.

The user specifies which channel to modify by supplying the submodule number and whether to modify PWM A or PWM B within that submodule.

Parameters

- base – PWM peripheral base address
- submodule – PWM submodule to configure
- pwmChannel – Channel to configure
- value – true: Supply a logic 1, false: Supply a logic 0.

```
static inline void PWM_SetPwmLdok(PWM_Type *base, uint8_t subModulesToUpdate, bool
                                  value)
```

Sets or clears the PWM LDOK bit on a single or multiple submodules.

Set LDOK bit to load buffered values into CTRL[PRSC] and the INIT, FRACVAL and VAL registers. The values are loaded immediately if `kPWM_ReloadImmediate` option was chosen during config. Else the values are loaded at the next PWM reload point. This function can issue the load command to multiple submodules at the same time.

Parameters

- base – PWM peripheral base address
- subModulesToUpdate – PWM submodules to update with buffered values. This is a logical OR of members of the enumeration `pwm_module_control_t`
- value – true: Set LDOK bit for the submodule list; false: Clear LDOK bit

```
static inline void PWM_SetPwmFaultState(PWM_Type *base, pwm_submodule_t submodule,
                                        pwm_channels_t pwmChannel, pwm_fault_state_t
                                        faultState)
```

Set PWM output fault status.

These bits determine the fault state for the PWM_A output in fault conditions and STOP mode. It may also define the output state in WAIT and DEBUG modes depending on the settings of CTRL2[WAITEN] and CTRL2[DBGEN]. This function can update PWM output fault status.

Parameters

- base – PWM peripheral base address
- submodule – PWM submodule to configure
- pwmChannel – Channel to configure

- `faultState` – PWM output fault status

```
static inline void PWM_SetupFaultDisableMap(PWM_Type *base, pwm_submodule_t subModule,
                                           pwm_channels_t pwmChannel,
                                           pwm_fault_channels_t pwm_fault_channels,
                                           uint16_t value)
```

Set PWM fault disable mapping.

Each of the four bits of this read/write field is one-to-one associated with the four FAULTx inputs of fault channel 0/1. The PWM output will be turned off if there is a logic 1 on an FAULTx input and a 1 in the corresponding bit of this field. A reset sets all bits in this field.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `pwmChannel` – PWM channel to configure
- `pwm_fault_channels` – PWM fault channel to configure
- `value` – Fault disable mapping mask value enumeration `pwm_fault_disable_t`

```
static inline void PWM_OutputEnable(PWM_Type *base, pwm_channels_t pwmChannel,
                                   pwm_submodule_t subModule)
```

Set PWM output enable.

This feature allows the user to enable the PWM Output. Recommend to invoke this API after PWM and fault configuration. But invoke this API before configure MCTRL register is okay, such as set LDOK or start timer.

Parameters

- `base` – PWM peripheral base address
- `pwmChannel` – PWM channel to configure
- `subModule` – PWM submodule to configure

```
static inline void PWM_OutputDisable(PWM_Type *base, pwm_channels_t pwmChannel,
                                    pwm_submodule_t subModule)
```

Set PWM output disable.

This feature allows the user to disable the PWM output. Recommend to invoke this API after PWM and fault configuration. But invoke this API before configure MCTRL register is okay, such as set LDOK or start timer.

Parameters

- `base` – PWM peripheral base address
- `pwmChannel` – PWM channel to configure
- `subModule` – PWM submodule to configure

```
uint8_t PWM_GetPwmChannelState(PWM_Type *base, pwm_submodule_t subModule,
                              pwm_channels_t pwmChannel)
```

Get the dutycycle value.

Parameters

- `base` – PWM peripheral base address
- `subModule` – PWM submodule to configure
- `pwmChannel` – PWM channel to configure

Returns

Current channel dutycycle value.

status_t PWM_SetOutputToIdle(PWM_Type *base, *pwm_channels_t* pwmChannel,
pwm_submodule_t subModule, bool idleStatus)

Set PWM output in idle status (high or low).

Note: This API should call after PWM_SetupPwm() APIs, and PWMX submodule is not supported.

Parameters

- base – PWM peripheral base address
- pwmChannel – PWM channel to configure
- subModule – PWM submodule to configure
- idleStatus – True: PWM output is high in idle status; false: PWM output is low in idle status.

Returns

kStatus_Fail if there was error setting up the signal; kStatus_Success if set output idle success

void PWM_SetClockMode(PWM_Type *base, *pwm_submodule_t* subModule,
pwm_clock_prescale_t prescaler)

Set the pwm submodule prescaler.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- prescaler – Set prescaler value

void PWM_SetPwmForceOutputToZero(PWM_Type *base, *pwm_submodule_t* subModule,
pwm_channels_t pwmChannel, bool forcetozero)

This function enables-disables the forcing of the output of a given eFlexPwm channel to logic 0.

Parameters

- base – PWM peripheral base address
- pwmChannel – PWM channel to configure
- subModule – PWM submodule to configure
- forcetozero – True: Enable the pwm force output to zero; False: Disable the pwm output resumes normal function.

void PWM_SetChannelOutput(PWM_Type *base, *pwm_submodule_t* subModule,
pwm_channels_t pwmChannel, *pwm_output_state_t* outputstate)

This function set the output state of the PWM pin as requested for the current cycle.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – PWM channel to configure
- outputstate – Set pwm output state, see *pwm_output_state_t*.


```
status_t PWM_SetPhaseDelay(PWM_Type *base, pwm_channels_t pwmChannel,
                          pwm_submodule_t subModule, uint16_t delayCycles)
```

This function set the phase delay from the master sync signal of submodule 0.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – PWM channel to configure
- delayCycles – Number of cycles delayed from submodule 0.

Returns

kStatus_Fail if the number of delay cycles is set larger than the period defined in submodule 0; kStatus_Success if set phase delay success

```
static inline void PWM_SetFilterSampleCount(PWM_Type *base, pwm_channels_t pwmChannel,
                                           pwm_submodule_t subModule, uint8_t
                                           filterSampleCount)
```

This function set the number of consecutive samples that must agree prior to the input filter.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – PWM channel to configure
- filterSampleCount – Number of consecutive samples.

```
static inline void PWM_SetFilterSamplePeriod(PWM_Type *base, pwm_channels_t pwmChannel,
                                           pwm_submodule_t subModule, uint8_t
                                           filterSamplePeriod)
```

This function set the sampling period of the fault pin input filter.

Parameters

- base – PWM peripheral base address
- subModule – PWM submodule to configure
- pwmChannel – PWM channel to configure
- filterSamplePeriod – Sampling period of input filter.

```
PWM_SUBMODULE_SWCONTROL_WIDTH
```

Number of bits per submodule for software output control

```
PWM_SUBMODULE_CHANNEL
```

Submodule channels include PWMA, PWMB, PWMX.

```
struct _pwm_signal_param
```

#include <fsl_pwm.h> Structure for the user to define the PWM signal characteristics.

Public Members

```
pwm_channels_t pwmChannel
```

PWM channel being configured; PWM A or PWM B

```
uint8_t dutyCyclePercent
```

PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...
100=always active signal (100% duty cycle)

pwm_level_select_t level

PWM output active level select

uint16_t deadtimeValue

The deadtime value; only used if channel pair is operating in complementary mode

pwm_fault_state_t faultState

PWM output fault status

bool pwmchannelenable

Enable PWM output

struct __pwm_config

#include <fsl_pwm.h> PWM config structure.

This structure holds the configuration settings for the PWM peripheral. To initialize this structure to reasonable defaults, call the PWM_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 enableDebugMode

true: PWM continues to run in debug mode; false: PWM is paused in debug mode

pwm_init_source_t initializationControl

Option to initialize the counter

pwm_clock_source_t clockSource

Clock source for the counter

pwm_clock_prescale_t prescale

Pre-scaler to divide down the clock

pwm_chnl_pair_operation_t pairOperation

Channel pair in independent or complementary mode

pwm_register_reload_t reloadLogic

PWM Reload logic setup

pwm_reload_source_select_t reloadSelect

Reload source select

pwm_load_frequency_t reloadFrequency

Specifies when to reload, used when user's choice is not immediate reload

pwm_force_output_trigger_t forceTrigger

Specify which signal will trigger a FORCE_OUT

struct __pwm_fault_input_filter_param

#include <fsl_pwm.h> Structure for the user to configure the fault input filter.

Public Members

uint8_t faultFilterCount

Fault filter count

uint8_t faultFilterPeriod

Fault filter period; value of 0 will bypass the filter

bool faultGlitchStretch

Fault Glitch Stretch Enable: A logic 1 means that input fault signals will be stretched to at least 2 IPBus clock cycles

struct __pwm_fault_param

#include <fsl_pwm.h> Structure is used to hold the parameters to configure a PWM fault.

Public Members

pwm_fault_clear_t faultClearingMode

Fault clearing mode to use

bool faultLevel

true: Logic 1 indicates fault; false: Logic 0 indicates fault

bool enableCombinationalPath

true: Combinational Path from fault input is enabled; false: No combination path is available

pwm_fault_recovery_mode_t recoverMode

Specify when to re-enable the PWM output

struct __pwm_input_capture_param

#include <fsl_pwm.h> Structure is used to hold parameters to configure the capture capability of a signal pin.

Public Members

bool captureInputSel

true: Use the edge counter signal as source false: Use the raw input signal from the pin as source

uint8_t edgeCompareValue

Compare value, used only if edge counter is used as source

pwm_input_capture_edge_t edge0

Specify which edge causes a capture for input circuitry 0

pwm_input_capture_edge_t edge1

Specify which edge causes a capture for input circuitry 1

bool enableOneShotCapture

true: Use one-shot capture mode; false: Use free-running capture mode

uint8_t fifoWatermark

Watermark level for capture FIFO. The capture flags in the status register will set if the word count in the FIFO is greater than this watermark level

2.49 RTWDOG: 32-bit Watchdog Timer

void RTWDOG_GetDefaultConfig(*rtwdog_config_t* *config)

Initializes the RTWDOG configuration structure.

This function initializes the RTWDOG configuration structure to default values. The default values are:

```
rtwdogConfig->enableRtwdog = true;
rtwdogConfig->clockSource = kRTWDOG_ClockSource1;
rtwdogConfig->prescaler = kRTWDOG_ClockPrescalerDivide1;
rtwdogConfig->workMode.enableWait = true;
rtwdogConfig->workMode.enableStop = false;
rtwdogConfig->workMode.enableDebug = false;
rtwdogConfig->testMode = kRTWDOG_TestModeDisabled;
rtwdogConfig->enableUpdate = true;
rtwdogConfig->enableInterrupt = false;
rtwdogConfig->enableWindowMode = false;
rtwdogConfig->windowValue = 0U;
rtwdogConfig->timeoutValue = 0xFFFFU;
```

See also:

rtwdog_config_t

Parameters

- config – Pointer to the RTWDOG configuration structure.

void RTWDOG_Init(RTWDOG_Type *base, const *rtwdog_config_t* *config)

Initializes the RTWDOG module.

This function initializes the RTWDOG. To reconfigure the RTWDOG without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
rtwdog_config_t config;
RTWDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
RTWDOG_Init(wdog_base,&config);
```

Parameters

- base – RTWDOG peripheral base address.
- config – The configuration of the RTWDOG.

void RTWDOG_Deinit(RTWDOG_Type *base)

De-initializes the RTWDOG module.

This function shuts down the RTWDOG. Ensure that the WDOG_CS.UPDATE is 1, which means that the register update is enabled.

Parameters

- base – RTWDOG peripheral base address.

static inline void RTWDOG_Enable(RTWDOG_Type *base)

Enables the RTWDOG module.

This function writes a value into the WDOG_CS register to enable the RTWDOG. The WDOG_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – RTWDOG peripheral base address.

```
static inline void RTWDOG_Disable(RTWDOG_Type *base)
```

Disables the RTWDOG module.

This function writes a value into the WDOG_CS register to disable the RTWDOG. The WDOG_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – RTWDOG peripheral base address

```
static inline void RTWDOG_EnableInterrupts(RTWDOG_Type *base, uint32_t mask)
```

Enables the RTWDOG interrupt.

This function writes a value into the WDOG_CS register to enable the RTWDOG interrupt. The WDOG_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – RTWDOG peripheral base address.
- `mask` – The interrupts to enable. The parameter can be a combination of the following source if defined:
 - `kRTWDOG_InterruptEnable`

```
static inline void RTWDOG_DisableInterrupts(RTWDOG_Type *base, uint32_t mask)
```

Disables the RTWDOG interrupt.

This function writes a value into the WDOG_CS register to disable the RTWDOG interrupt. The WDOG_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – RTWDOG peripheral base address.
- `mask` – The interrupts to disabled. The parameter can be a combination of the following source if defined:
 - `kRTWDOG_InterruptEnable`

```
static inline uint32_t RTWDOG_GetStatusFlags(RTWDOG_Type *base)
```

Gets the RTWDOG all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;
status = RTWDOG_GetStatusFlags(wdog_base) & kRTWDOG_RunningFlag;
```

See also:

`_rtwdog_status_flags_t`

- `true`: related status flag has been set.
- `false`: related status flag is not set.

Parameters

- `base` – RTWDOG peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

```
static inline void RTWDOG_EnableWindowMode(RTWDOG_Type *base, bool enable)
```

Enables/disables the window mode.

Parameters

- base – RTWDOG peripheral base address.
- enable – Enables(true) or disables(false) the feature.

```
static inline uint32_t RTWDOG_CountToMsec(RTWDOG_Type *base, uint32_t count, uint32_t  
clockFreqInHz)
```

Converts raw count value to millisecond.

Note that if the clock frequency is too high the timeout period can be less than 1 ms. In this case this api will return 0 value.

Parameters

- base – RTWDOG peripheral base address.
- count – Raw count value.
- clockFreqInHz – The frequency of the clock source RTWDOG uses.

Returns

Return converted time. Will return 0 if result is larger than 0xFFFFFFFF.

```
void RTWDOG_ClearStatusFlags(RTWDOG_Type *base, uint32_t mask)
```

Clears the RTWDOG flag.

This function clears the RTWDOG status flag.

Example to clear an interrupt flag:

```
RTWDOG_ClearStatusFlags(wdog_base,kRTWDOG_InterruptFlag);
```

Parameters

- base – RTWDOG peripheral base address.
- mask – The status flags to clear. The parameter can be any combination of the following values:
 - kRTWDOG_InterruptFlag

```
static inline void RTWDOG_SetTimeoutValue(RTWDOG_Type *base, uint16_t timeoutCount)
```

Sets the RTWDOG timeout value.

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – RTWDOG peripheral base address
- timeoutCount – RTWDOG timeout value, count of RTWDOG clock ticks.

```
static inline void RTWDOG_SetWindowValue(RTWDOG_Type *base, uint16_t windowValue)
```

Sets the RTWDOG window value.

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – RTWDOG peripheral base address.
- windowValue – RTWDOG window value.

`__STATIC_FORCEINLINE void RTWDOG_Unlock (RTWDOG_Type *base)`

Unlocks the RTWDOG register written.

This function unlocks the RTWDOG register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – RTWDOG peripheral base address

`static inline void RTWDOG_Refresh(RTWDOG_Type *base)`

Refreshes the RTWDOG timer.

This function feeds the RTWDOG. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – RTWDOG peripheral base address

`static inline uint32_t RTWDOG_GetCounterValue(RTWDOG_Type *base)`

Gets the RTWDOG counter value.

This function gets the RTWDOG counter value.

Parameters

- base – RTWDOG peripheral base address.

Returns

Current RTWDOG counter value.

`WDOG_FIRST_WORD_OF_UNLOCK`

First word of unlock sequence

`WDOG_SECOND_WORD_OF_UNLOCK`

Second word of unlock sequence

`WDOG_FIRST_WORD_OF_REFRESH`

First word of refresh sequence

`WDOG_SECOND_WORD_OF_REFRESH`

Second word of refresh sequence

`FSL_RTWDOG_DRIVER_VERSION`

RTWDOG driver version.

`enum _rtwdog_clock_source`

Describes RTWDOG clock source.

Values:

enumerator `kRTWDOG_ClockSource0`

Clock source 0

enumerator `kRTWDOG_ClockSource1`

Clock source 1

enumerator `kRTWDOG_ClockSource2`

Clock source 2

enumerator `kRTWDOG_ClockSource3`

Clock source 3

enum *_rtwdog_clock_prescaler*

Describes the selection of the clock prescaler.

Values:

enumerator *kRTWDOG_ClockPrescalerDivide1*

Divided by 1

enumerator *kRTWDOG_ClockPrescalerDivide256*

Divided by 256

enum *_rtwdog_test_mode*

Describes RTWDOG test mode.

Values:

enumerator *kRTWDOG_TestModeDisabled*

Test Mode disabled

enumerator *kRTWDOG_UserModeEnabled*

User Mode enabled

enumerator *kRTWDOG_LowByteTest*

Test Mode enabled, only low byte is used

enumerator *kRTWDOG_HighByteTest*

Test Mode enabled, only high byte is used

enum *_rtwdog_interrupt_enable_t*

RTWDOG interrupt configuration structure.

This structure contains the settings for all of the RTWDOG interrupt configurations.

Values:

enumerator *kRTWDOG_InterruptEnable*

Interrupt is generated before forcing a reset

enum *_rtwdog_status_flags_t*

RTWDOG status flags.

This structure contains the RTWDOG status flags for use in the RTWDOG functions.

Values:

enumerator *kRTWDOG_RunningFlag*

Running flag, set when RTWDOG is enabled

enumerator *kRTWDOG_InterruptFlag*

Interrupt flag, set when interrupt occurs

typedef enum *_rtwdog_clock_source* *rtwdog_clock_source_t*

Describes RTWDOG clock source.

typedef enum *_rtwdog_clock_prescaler* *rtwdog_clock_prescaler_t*

Describes the selection of the clock prescaler.

typedef struct *_rtwdog_work_mode* *rtwdog_work_mode_t*

Defines RTWDOG work mode.

typedef enum *_rtwdog_test_mode* *rtwdog_test_mode_t*

Describes RTWDOG test mode.

typedef struct *_rtwdog_config* *rtwdog_config_t*

Describes RTWDOG configuration structure.

struct `_rtwdog_work_mode`

#include <fsl_rtwdog.h> Defines RTWDOG work mode.

Public Members

bool `enableWait`

Enables or disables RTWDOG in wait mode

bool `enableStop`

Enables or disables RTWDOG in stop mode

bool `enableDebug`

Enables or disables RTWDOG in debug mode

struct `_rtwdog_config`

#include <fsl_rtwdog.h> Describes RTWDOG configuration structure.

Public Members

bool `enableRtwdog`

Enables or disables RTWDOG

rtwdog_clock_source_t `clockSource`

Clock source select

rtwdog_clock_prescaler_t `prescaler`

Clock prescaler value

rtwdog_work_mode_t `workMode`

Configures RTWDOG work mode in debug stop and wait mode

rtwdog_test_mode_t `testMode`

Configures RTWDOG test mode

bool `enableUpdate`

Update write-once register enable

bool `enableInterrupt`

Enables or disables RTWDOG interrupt

bool `enableWindowMode`

Enables or disables RTWDOG window mode

uint16_t `windowValue`

Window value

uint16_t `timeoutValue`

Timeout value

2.50 SAI: Serial Audio Interface

2.51 SAI Driver

`void SAI_Init(I2S_Type *base)`

Initializes the SAI peripheral.

This API gates the SAI clock. The SAI module can't operate unless SAI_Init is called to enable the clock.

Parameters

- base – SAI base pointer.

`void SAI_Deinit(I2S_Type *base)`

De-initializes the SAI peripheral.

This API gates the SAI clock. The SAI module can't operate unless SAI_TxInit or SAI_RxInit is called to enable the clock.

Parameters

- base – SAI base pointer.

`void SAI_TxReset(I2S_Type *base)`

Resets the SAI Tx.

This function enables the software reset and FIFO reset of SAI Tx. After reset, clear the reset bit.

Parameters

- base – SAI base pointer

`void SAI_RxReset(I2S_Type *base)`

Resets the SAI Rx.

This function enables the software reset and FIFO reset of SAI Rx. After reset, clear the reset bit.

Parameters

- base – SAI base pointer

`void SAI_TxEnable(I2S_Type *base, bool enable)`

Enables/disables the SAI Tx.

Parameters

- base – SAI base pointer.
- enable – True means enable SAI Tx, false means disable.

`void SAI_RxEnable(I2S_Type *base, bool enable)`

Enables/disables the SAI Rx.

Parameters

- base – SAI base pointer.
- enable – True means enable SAI Rx, false means disable.

`static inline void SAI_TxSetBitClockDirection(I2S_Type *base, sai_master_slave_t masterSlave)`

Set Rx bit clock direction.

Select bit clock direction, master or slave.

Parameters

- base – SAI base pointer.
- masterSlave – reference sai_master_slave_t.

```
static inline void SAI_RxSetBitClockDirection(I2S_Type *base, sai_master_slave_t masterSlave)
```

Set Rx bit clock direction.

Select bit clock direction, master or slave.

Parameters

- base – SAI base pointer.
- masterSlave – reference sai_master_slave_t.

```
static inline void SAI_RxSetFrameSyncDirection(I2S_Type *base, sai_master_slave_t masterSlave)
```

Set Rx frame sync direction.

Select frame sync direction, master or slave.

Parameters

- base – SAI base pointer.
- masterSlave – reference sai_master_slave_t.

```
static inline void SAI_TxSetFrameSyncDirection(I2S_Type *base, sai_master_slave_t masterSlave)
```

Set Tx frame sync direction.

Select frame sync direction, master or slave.

Parameters

- base – SAI base pointer.
- masterSlave – reference sai_master_slave_t.

```
void SAI_TxSetBitClockRate(I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)
```

Transmitter bit clock rate configurations.

Parameters

- base – SAI base pointer.
- sourceClockHz – Bit clock source frequency.
- sampleRate – Audio data sample rate.
- bitWidth – Audio data bitWidth.
- channelNumbers – Audio channel numbers.

```
void SAI_RxSetBitClockRate(I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)
```

Receiver bit clock rate configurations.

Parameters

- base – SAI base pointer.
- sourceClockHz – Bit clock source frequency.
- sampleRate – Audio data sample rate.
- bitWidth – Audio data bitWidth.
- channelNumbers – Audio channel numbers.

```
void SAI_TxSetBitclockConfig(I2S_Type *base, sai_master_slave_t masterSlave, sai_bit_clock_t *config)
```

Transmitter Bit clock configurations.

Parameters

- base – SAI base pointer.
- masterSlave – master or slave.
- config – bit clock other configurations, can be NULL in slave mode.

```
void SAI_RxSetBitclockConfig(I2S_Type *base, sai_master_slave_t masterSlave, sai_bit_clock_t *config)
```

Receiver Bit clock configurations.

Parameters

- base – SAI base pointer.
- masterSlave – master or slave.
- config – bit clock other configurations, can be NULL in slave mode.

```
void SAI_SetMasterClockConfig(I2S_Type *base, sai_master_clock_t *config)
```

Master clock configurations.

Parameters

- base – SAI base pointer.
- config – master clock configurations.

```
void SAI_TxSetFifoConfig(I2S_Type *base, sai_fifo_t *config)
```

SAI transmitter fifo configurations.

Parameters

- base – SAI base pointer.
- config – fifo configurations.

```
void SAI_RxSetFifoConfig(I2S_Type *base, sai_fifo_t *config)
```

SAI receiver fifo configurations.

Parameters

- base – SAI base pointer.
- config – fifo configurations.

```
void SAI_TxSetFrameSyncConfig(I2S_Type *base, sai_master_slave_t masterSlave, sai_frame_sync_t *config)
```

SAI transmitter Frame sync configurations.

Parameters

- base – SAI base pointer.
- masterSlave – master or slave.
- config – frame sync configurations, can be NULL in slave mode.

```
void SAI_RxSetFrameSyncConfig(I2S_Type *base, sai_master_slave_t masterSlave, sai_frame_sync_t *config)
```

SAI receiver Frame sync configurations.

Parameters

- base – SAI base pointer.
- masterSlave – master or slave.
- config – frame sync configurations, can be NULL in slave mode.

void SAI_TxSetSerialDataConfig(I2S_Type *base, *sai_serial_data_t* *config)
SAI transmitter Serial data configurations.

Parameters

- base – SAI base pointer.
- config – serial data configurations.

void SAI_RxSetSerialDataConfig(I2S_Type *base, *sai_serial_data_t* *config)
SAI receiver Serial data configurations.

Parameters

- base – SAI base pointer.
- config – serial data configurations.

void SAI_TxSetConfig(I2S_Type *base, *sai_transceiver_t* *config)
SAI transmitter configurations.

Parameters

- base – SAI base pointer.
- config – transmitter configurations.

void SAI_RxSetConfig(I2S_Type *base, *sai_transceiver_t* *config)
SAI receiver configurations.

Parameters

- base – SAI base pointer.
- config – receiver configurations.

void SAI_GetClassicI2SConfig(*sai_transceiver_t* *config, *sai_word_width_t* bitWidth,
sai_mono_stereo_t mode, uint32_t saiChannelMask)

Get classic I2S mode configurations.

Parameters

- config – transceiver configurations.
- bitWidth – audio data bitWidth.
- mode – audio data channel.
- saiChannelMask – mask value of the channel to be enable.

void SAI_GetLeftJustifiedConfig(*sai_transceiver_t* *config, *sai_word_width_t* bitWidth,
sai_mono_stereo_t mode, uint32_t saiChannelMask)

Get left justified mode configurations.

Parameters

- config – transceiver configurations.
- bitWidth – audio data bitWidth.
- mode – audio data channel.
- saiChannelMask – mask value of the channel to be enable.

void SAI_GetRightJustifiedConfig(*sai_transceiver_t* *config, *sai_word_width_t* bitWidth,
sai_mono_stereo_t mode, uint32_t saiChannelMask)

Get right justified mode configurations.

Parameters

- config – transceiver configurations.

- bitWidth – audio data bitWidth.
- mode – audio data channel.
- saiChannelMask – mask value of the channel to be enable.

```
void SAI_GetTDMConfig(sai_transceiver_t *config, sai_frame_sync_len_t frameSyncWidth,  
                    sai_word_width_t bitWidth, uint32_t dataWordNum, uint32_t  
                    saiChannelMask)
```

Get TDM mode configurations.

Parameters

- config – transceiver configurations.
- frameSyncWidth – length of frame sync.
- bitWidth – audio data word width.
- dataWordNum – word number in one frame.
- saiChannelMask – mask value of the channel to be enable.

```
void SAI_GetDSPConfig(sai_transceiver_t *config, sai_frame_sync_len_t frameSyncWidth,  
                    sai_word_width_t bitWidth, sai_mono_stereo_t mode, uint32_t  
                    saiChannelMask)
```

Get DSP mode configurations.

DSP/PCM MODE B configuration flow for TX. RX is similiar but uses SAI_RxSetConfig instead of SAI_TxSetConfig:

```
SAI_GetDSPConfig(config, kSAI_FrameSyncLenOneBitClk, bitWidth, kSAI_Stereo, channelMask)  
SAI_TxSetConfig(base, config)
```

Note: DSP mode is also called PCM mode which support MODE A and MODE B, DSP/PCM MODE A configuration flow. RX is similiar but uses SAI_RxSetConfig instead of SAI_TxSetConfig:

```
SAI_GetDSPConfig(config, kSAI_FrameSyncLenOneBitClk, bitWidth, kSAI_Stereo, channelMask)  
config->frameSync.frameSyncEarly = true;  
SAI_TxSetConfig(base, config)
```

Parameters

- config – transceiver configurations.
- frameSyncWidth – length of frame sync.
- bitWidth – audio data bitWidth.
- mode – audio data channel.
- saiChannelMask – mask value of the channel to be enable.

```
static inline uint32_t SAI_TxGetStatusFlag(I2S_Type *base)
```

Gets the SAI Tx status flag state.

Parameters

- base – SAI base pointer

Returns

SAI Tx status flag value. Use the Status Mask to get the status value needed.

```
static inline void SAI_TxClearStatusFlags(I2S_Type *base, uint32_t mask)
```

Clears the SAI Tx status flag state.

Parameters

- base – SAI base pointer
- mask – State mask. It can be a combination of the following source if defined:
 - kSAI_WordStartFlag
 - kSAI_SyncErrorFlag
 - kSAI_FIFOErrorFlag

```
static inline uint32_t SAI_RxGetStatusFlag(I2S_Type *base)
```

Gets the SAI Tx status flag state.

Parameters

- base – SAI base pointer

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

```
static inline void SAI_RxClearStatusFlags(I2S_Type *base, uint32_t mask)
```

Clears the SAI Rx status flag state.

Parameters

- base – SAI base pointer
- mask – State mask. It can be a combination of the following sources if defined.
 - kSAI_WordStartFlag
 - kSAI_SyncErrorFlag
 - kSAI_FIFOErrorFlag

```
void SAI_TxSoftwareReset(I2S_Type *base, sai_reset_type_t resetType)
```

Do software reset or FIFO reset .

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Tx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like TCR1~TCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Parameters

- base – SAI base pointer
- resetType – Reset type, FIFO reset or software reset

```
void SAI_RxSoftwareReset(I2S_Type *base, sai_reset_type_t resetType)
```

Do software reset or FIFO reset .

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Rx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like RCR1~RCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Parameters

- base – SAI base pointer
- resetType – Reset type, FIFO reset or software reset

`void SAI_TxSetChannelFIFOMask(I2S_Type *base, uint8_t mask)`

Set the Tx channel FIFO enable mask.

Parameters

- `base` – SAI base pointer
- `mask` – Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

`void SAI_RxSetChannelFIFOMask(I2S_Type *base, uint8_t mask)`

Set the Rx channel FIFO enable mask.

Parameters

- `base` – SAI base pointer
- `mask` – Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

`void SAI_TxSetDataOrder(I2S_Type *base, sai_data_order_t order)`

Set the Tx data order.

Parameters

- `base` – SAI base pointer
- `order` – Data order MSB or LSB

`void SAI_RxSetDataOrder(I2S_Type *base, sai_data_order_t order)`

Set the Rx data order.

Parameters

- `base` – SAI base pointer
- `order` – Data order MSB or LSB

`void SAI_TxSetBitClockPolarity(I2S_Type *base, sai_clock_polarity_t polarity)`

Set the Tx data order.

Parameters

- `base` – SAI base pointer
- `polarity` –

`void SAI_RxSetBitClockPolarity(I2S_Type *base, sai_clock_polarity_t polarity)`

Set the Rx data order.

Parameters

- `base` – SAI base pointer
- `polarity` –

`void SAI_TxSetFrameSyncPolarity(I2S_Type *base, sai_clock_polarity_t polarity)`

Set the Tx data order.

Parameters

- `base` – SAI base pointer
- `polarity` –

`void SAI_RxSetFrameSyncPolarity(I2S_Type *base, sai_clock_polarity_t polarity)`

Set the Rx data order.

Parameters

- `base` – SAI base pointer

- polarity –

void SAI_TxSetFIFOPacking(I2S_Type *base, *sai_fifo_packing_t* pack)

Set Tx FIFO packing feature.

Parameters

- base – SAI base pointer.
- pack – FIFO pack type. It is element of *sai_fifo_packing_t*.

void SAI_RxSetFIFOPacking(I2S_Type *base, *sai_fifo_packing_t* pack)

Set Rx FIFO packing feature.

Parameters

- base – SAI base pointer.
- pack – FIFO pack type. It is element of *sai_fifo_packing_t*.

static inline void SAI_TxSetFIFOErrorContinue(I2S_Type *base, bool isEnabled)

Set Tx FIFO error continue.

FIFO error continue mode means SAI will keep running while FIFO error occurred. If this feature not enabled, SAI will hang and users need to clear FEF flag in TCSR register.

Parameters

- base – SAI base pointer.
- isEnabled – Is FIFO error continue enabled, true means enable, false means disable.

static inline void SAI_RxSetFIFOErrorContinue(I2S_Type *base, bool isEnabled)

Set Rx FIFO error continue.

FIFO error continue mode means SAI will keep running while FIFO error occurred. If this feature not enabled, SAI will hang and users need to clear FEF flag in RCSR register.

Parameters

- base – SAI base pointer.
- isEnabled – Is FIFO error continue enabled, true means enable, false means disable.

static inline void SAI_TxEnableInterrupts(I2S_Type *base, uint32_t mask)

Enables the SAI Tx interrupt requests.

Parameters

- base – SAI base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSAI_WordStartInterruptEnable
 - kSAI_SyncErrorInterruptEnable
 - kSAI_FIFOWarningInterruptEnable
 - kSAI_FIFORequestInterruptEnable
 - kSAI_FIFOErrorInterruptEnable

static inline void SAI_RxEnableInterrupts(I2S_Type *base, uint32_t mask)

Enables the SAI Rx interrupt requests.

Parameters

- base – SAI base pointer

- **mask** – interrupt source The parameter can be a combination of the following sources if defined.
 - `kSAI_WordStartInterruptEnable`
 - `kSAI_SyncErrorInterruptEnable`
 - `kSAI_FIFOWarningInterruptEnable`
 - `kSAI_FIFORequestInterruptEnable`
 - `kSAI_FIFOErrorInterruptEnable`

`static inline void SAI_TxDisableInterrupts(I2S_Type *base, uint32_t mask)`

Disables the SAI Tx interrupt requests.

Parameters

- **base** – SAI base pointer
- **mask** – interrupt source The parameter can be a combination of the following sources if defined.
 - `kSAI_WordStartInterruptEnable`
 - `kSAI_SyncErrorInterruptEnable`
 - `kSAI_FIFOWarningInterruptEnable`
 - `kSAI_FIFORequestInterruptEnable`
 - `kSAI_FIFOErrorInterruptEnable`

`static inline void SAI_RxDisableInterrupts(I2S_Type *base, uint32_t mask)`

Disables the SAI Rx interrupt requests.

Parameters

- **base** – SAI base pointer
- **mask** – interrupt source The parameter can be a combination of the following sources if defined.
 - `kSAI_WordStartInterruptEnable`
 - `kSAI_SyncErrorInterruptEnable`
 - `kSAI_FIFOWarningInterruptEnable`
 - `kSAI_FIFORequestInterruptEnable`
 - `kSAI_FIFOErrorInterruptEnable`

`static inline void SAI_TxEnableDMA(I2S_Type *base, uint32_t mask, bool enable)`

Enables/disables the SAI Tx DMA requests.

Parameters

- **base** – SAI base pointer
- **mask** – DMA source The parameter can be combination of the following sources if defined.
 - `kSAI_FIFOWarningDMAEnable`
 - `kSAI_FIFORequestDMAEnable`
- **enable** – True means enable DMA, false means disable DMA.

```
static inline void SAI_RxEnableDMA(I2S_Type *base, uint32_t mask, bool enable)
```

Enables/disables the SAI Rx DMA requests.

Parameters

- base – SAI base pointer
- mask – DMA source The parameter can be a combination of the following sources if defined.
 - kSAI_FIFOWarningDMAEnable
 - kSAI_FIFORequestDMAEnable
- enable – True means enable DMA, false means disable DMA.

```
static inline uintptr_t SAI_TxGetDataRegisterAddress(I2S_Type *base, uint32_t channel)
```

Gets the SAI Tx data register address.

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

- base – SAI base pointer.
- channel – Which data channel used.

Returns

data register address.

```
static inline uintptr_t SAI_RxGetDataRegisterAddress(I2S_Type *base, uint32_t channel)
```

Gets the SAI Rx data register address.

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

- base – SAI base pointer.
- channel – Which data channel used.

Returns

data register address.

```
void SAI_WriteBlocking(I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer,  
                      uint32_t size)
```

Sends data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

```
void SAI_WriteMultiChannelBlocking(I2S_Type *base, uint32_t channel, uint32_t channelMask,  
                                  uint32_t bitWidth, uint8_t *buffer, uint32_t size)
```

Sends data to multi channel using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- channelMask – channel mask.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

```
static inline void SAI_WriteData(I2S_Type *base, uint32_t channel, uint32_t data)
```

Writes data into SAI FIFO.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- data – Data needs to be written.

```
void SAI_ReadBlocking(I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer,  
                     uint32_t size)
```

Receives data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be read.
- size – Bytes to be read.

```
void SAI_ReadMultiChannelBlocking(I2S_Type *base, uint32_t channel, uint32_t channelMask,  
                                 uint32_t bitWidth, uint8_t *buffer, uint32_t size)
```

Receives multi channel data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- channelMask – channel mask.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be read.
- size – Bytes to be read.

```
static inline uint32_t SAI_ReadData(I2S_Type *base, uint32_t channel)
```

Reads data from the SAI FIFO.

Parameters

- base – SAI base pointer.
- channel – Data channel used.

Returns

Data in SAI FIFO.

```
void SAI_TransferTxCreateHandle(I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t  
callback, void *userData)
```

Initializes the SAI Tx handle.

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SAI base pointer
- handle – SAI handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function

```
void SAI_TransferRxCreateHandle(I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t  
callback, void *userData)
```

Initializes the SAI Rx handle.

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function.

```
void SAI_TransferTxSetConfig(I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)  
SAI transmitter transfer configurations.
```

This function initializes the Tx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.
- config – transmitter configurations.

```
void SAI_TransferRxSetConfig(I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)  
SAI receiver transfer configurations.
```

This function initializes the Rx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.

- config – receiver configurations.

status_t SAI_TransferSendNonBlocking(I2S_Type *base, *sai_handle_t* *handle, *sai_transfer_t* *xfer)

Performs an interrupt non-blocking send transfer on SAI.

Note: This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

- base – SAI base pointer.
- handle – Pointer to the *sai_handle_t* structure which stores the transfer state.
- xfer – Pointer to the *sai_transfer_t* structure.

Return values

- kStatus_Success – Successfully started the data receive.
- kStatus_SAI_TxBusy – Previous receive still not finished.
- kStatus_InvalidArgument – The input parameter is invalid.

status_t SAI_TransferReceiveNonBlocking(I2S_Type *base, *sai_handle_t* *handle, *sai_transfer_t* *xfer)

Performs an interrupt non-blocking receive transfer on SAI.

Note: This API returns immediately after the transfer initiates. Call the SAI_RxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

- base – SAI base pointer
- handle – Pointer to the *sai_handle_t* structure which stores the transfer state.
- xfer – Pointer to the *sai_transfer_t* structure.

Return values

- kStatus_Success – Successfully started the data receive.
- kStatus_SAI_RxBusy – Previous receive still not finished.
- kStatus_InvalidArgument – The input parameter is invalid.

status_t SAI_TransferGetSendCount(I2S_Type *base, *sai_handle_t* *handle, *size_t* *count)

Gets a set byte count.

Parameters

- base – SAI base pointer.
- handle – Pointer to the *sai_handle_t* structure which stores the transfer state.
- count – Bytes count sent.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`status_t SAI_TransferGetReceiveCount(I2S_Type *base, sai_handle_t *handle, size_t *count)`

Gets a received byte count.

Parameters

- `base` – SAI base pointer.
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.
- `count` – Bytes count received.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`void SAI_TransferAbortSend(I2S_Type *base, sai_handle_t *handle)`

Aborts the current send.

Note: This API can be called any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – SAI base pointer.
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.

`void SAI_TransferAbortReceive(I2S_Type *base, sai_handle_t *handle)`

Aborts the current IRQ receive.

Note: This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – SAI base pointer
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.

`void SAI_TransferTerminateSend(I2S_Type *base, sai_handle_t *handle)`

Terminate all SAI send.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call `SAI_TransferAbortSend`.

Parameters

- `base` – SAI base pointer.
- `handle` – SAI eDMA handle pointer.

void SAI_TransferTerminateReceive(I2S_Type *base, sai_handle_t *handle)

Terminate all SAI receive.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceive.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferTxHandleIRQ(I2S_Type *base, sai_handle_t *handle)

Tx interrupt handler.

Parameters

- base – SAI base pointer.
- handle – Pointer to the sai_handle_t structure.

void SAI_TransferRxHandleIRQ(I2S_Type *base, sai_handle_t *handle)

Tx interrupt handler.

Parameters

- base – SAI base pointer.
- handle – Pointer to the sai_handle_t structure.

void SAI_DriverIRQHandler(uint32_t instance)

SAI driver IRQ handler common entry.

This function provides the common IRQ request entry for SAI.

Parameters

- instance – SAI instance.

FSL_SAI_DRIVER_VERSION

Version 2.4.10

_sai_status_t, SAI return status.

Values:

enumerator kStatus_SAI_TxBusy
SAI Tx is busy.

enumerator kStatus_SAI_RxBusy
SAI Rx is busy.

enumerator kStatus_SAI_TxError
SAI Tx FIFO error.

enumerator kStatus_SAI_RxError
SAI Rx FIFO error.

enumerator kStatus_SAI_QueueFull
SAI transfer queue is full.

enumerator kStatus_SAI_TxIdle
SAI Tx is idle

enumerator kStatus_SAI_RxIdle
SAI Rx is idle

`_sai_channel_mask`, sai channel mask value, actual channel numbers is depend soc specific
Values:

enumerator `kSAI_Channel0Mask`
channel 0 mask value

enumerator `kSAI_Channel1Mask`
channel 1 mask value

enumerator `kSAI_Channel2Mask`
channel 2 mask value

enumerator `kSAI_Channel3Mask`
channel 3 mask value

enumerator `kSAI_Channel4Mask`
channel 4 mask value

enumerator `kSAI_Channel5Mask`
channel 5 mask value

enumerator `kSAI_Channel6Mask`
channel 6 mask value

enumerator `kSAI_Channel7Mask`
channel 7 mask value

enum `_sai_protocol`

Define the SAI bus type.

Values:

enumerator `kSAI_BusLeftJustified`
Uses left justified format.

enumerator `kSAI_BusRightJustified`
Uses right justified format.

enumerator `kSAI_BusI2S`
Uses I2S format.

enumerator `kSAI_BusPCMA`
Uses I2S PCM A format.

enumerator `kSAI_BusPCMB`
Uses I2S PCM B format.

enum `_sai_master_slave`

Master or slave mode.

Values:

enumerator `kSAI_Master`
Master mode include bclk and frame sync

enumerator `kSAI_Slave`
Slave mode include bclk and frame sync

enumerator `kSAI_Bclk_Master_FrameSync_Slave`
bclk in master mode, frame sync in slave mode

enumerator kSAI_Bclk_Slave_FrameSync_Master
bclk in slave mode, frame sync in master mode

enum _sai_mono_stereo
Mono or stereo audio format.

Values:

enumerator kSAI_Stereo
Stereo sound.

enumerator kSAI_MonoRight
Only Right channel have sound.

enumerator kSAI_MonoLeft
Only left channel have sound.

enum _sai_data_order
SAI data order, MSB or LSB.

Values:

enumerator kSAI_DataLSB
LSB bit transferred first

enumerator kSAI_DataMSB
MSB bit transferred first

enum _sai_clock_polarity
SAI clock polarity, active high or low.

Values:

enumerator kSAI_PolarityActiveHigh
Drive outputs on rising edge

enumerator kSAI_PolarityActiveLow
Drive outputs on falling edge

enumerator kSAI_SampleOnFallingEdge
Sample inputs on falling edge

enumerator kSAI_SampleOnRisingEdge
Sample inputs on rising edge

enum _sai_sync_mode
Synchronous or asynchronous mode.

Values:

enumerator kSAI_ModeAsync
Asynchronous mode

enumerator kSAI_ModeSync
Synchronous mode (with receiver or transmit)

enumerator kSAI_ModeSyncWithOtherTx
Synchronous with another SAI transmit

enumerator kSAI_ModeSyncWithOtherRx
Synchronous with another SAI receiver

enum _sai_bclk_source

Bit clock source.

Values:

enumerator kSAI_BclkSourceBusclk

Bit clock using bus clock

enumerator kSAI_BclkSourceMclkOption1

Bit clock MCLK option 1

enumerator kSAI_BclkSourceMclkOption2

Bit clock MCLK option2

enumerator kSAI_BclkSourceMclkOption3

Bit clock MCLK option3

enumerator kSAI_BclkSourceMclkDiv

Bit clock using master clock divider

enumerator kSAI_BclkSourceOtherSai0

Bit clock from other SAI device

enumerator kSAI_BclkSourceOtherSai1

Bit clock from other SAI device

_sai_interrupt_enable_t, The SAI interrupt enable flag

Values:

enumerator kSAI_WordStartInterruptEnable

Word start flag, means the first word in a frame detected

enumerator kSAI_SyncErrorInterruptEnable

Sync error flag, means the sync error is detected

enumerator kSAI_FIFOWarningInterruptEnable

FIFO warning flag, means the FIFO is empty

enumerator kSAI_FIFOErrorInterruptEnable

FIFO error flag

enumerator kSAI_FIFORequestInterruptEnable

FIFO request, means reached watermark

_sai_dma_enable_t, The DMA request sources

Values:

enumerator kSAI_FIFOWarningDMAEnable

FIFO warning caused by the DMA request

enumerator kSAI_FIFORequestDMAEnable

FIFO request caused by the DMA request

_sai_flags, The SAI status flag

Values:

enumerator kSAI_WordStartFlag

Word start flag, means the first word in a frame detected

enumerator kSAI_SyncErrorFlag

Sync error flag, means the sync error is detected

enumerator kSAI_FIFOErrorFlag

FIFO error flag

enumerator kSAI_FIFORequestFlag

FIFO request flag.

enumerator kSAI_FIFOWarningFlag

FIFO warning flag

enum _sai_reset_type

The reset type.

Values:

enumerator kSAI_ResetTypeSoftware

Software reset, reset the logic state

enumerator kSAI_ResetTypeFIFO

FIFO reset, reset the FIFO read and write pointer

enumerator kSAI_ResetAll

All reset.

enum _sai_fifo_packing

The SAI packing mode The mode includes 8 bit and 16 bit packing.

Values:

enumerator kSAI_FifoPackingDisabled

Packing disabled

enumerator kSAI_FifoPacking8bit

8 bit packing enabled

enumerator kSAI_FifoPacking16bit

16bit packing enabled

enum _sai_sample_rate

Audio sample rate.

Values:

enumerator kSAI_SampleRate8KHz

Sample rate 8000 Hz

enumerator kSAI_SampleRate11025Hz

Sample rate 11025 Hz

enumerator kSAI_SampleRate12KHz

Sample rate 12000 Hz

enumerator kSAI_SampleRate16KHz

Sample rate 16000 Hz

enumerator kSAI_SampleRate22050Hz

Sample rate 22050 Hz

enumerator kSAI_SampleRate24KHz

Sample rate 24000 Hz

enumerator kSAI_SampleRate32KHz

Sample rate 32000 Hz

enumerator kSAI_SampleRate44100Hz

Sample rate 44100 Hz

enumerator kSAI_SampleRate48KHz

Sample rate 48000 Hz

enumerator kSAI_SampleRate96KHz

Sample rate 96000 Hz

enumerator kSAI_SampleRate192KHz

Sample rate 192000 Hz

enumerator kSAI_SampleRate384KHz

Sample rate 384000 Hz

enum _sai_word_width

Audio word width.

Values:

enumerator kSAI_WordWidth8bits

Audio data width 8 bits

enumerator kSAI_WordWidth16bits

Audio data width 16 bits

enumerator kSAI_WordWidth24bits

Audio data width 24 bits

enumerator kSAI_WordWidth32bits

Audio data width 32 bits

enum _sai_data_pin_state

sai data pin state definition

Values:

enumerator kSAI_DataPinStateTriState

transmit data pins are tri-stated when slots are masked or channels are disabled

enumerator kSAI_DataPinStateOutputZero

transmit data pins are never tri-stated and will output zero when slots are masked or channel disabled

enum _sai_fifo_combine

sai fifo combine mode definition

Values:

enumerator kSAI_FifoCombineDisabled

sai TX/RX fifo combine mode disabled

enumerator kSAI_FifoCombineModeEnabledOnRead

sai TX fifo combine mode enabled on FIFO reads

enumerator kSAI_FifoCombineModeEnabledOnWrite

sai TX fifo combine mode enabled on FIFO write

enumerator kSAI_RxFifoCombineModeEnabledOnWrite

sai RX fifo combine mode enabled on FIFO write

enumerator kSAI_RXFifoCombineModeEnabledOnRead
sai RX fifo combine mode enabled on FIFO reads

enumerator kSAI_FifoCombineModeEnabledOnReadWrite
sai TX/RX fifo combined mode enabled on FIFO read/writes

enum __sai_transceiver_type
sai transceiver type

Values:

enumerator kSAI_Transmitter
sai transmitter

enumerator kSAI_Receiver
sai receiver

enum __sai_frame_sync_len
sai frame sync len

Values:

enumerator kSAI_FrameSyncLenOneBitClk
1 bit clock frame sync len for DSP mode

enumerator kSAI_FrameSyncLenPerWordWidth
Frame sync length decided by word width

typedef enum __sai_protocol sai_protocol_t
Define the SAI bus type.

typedef enum __sai_master_slave sai_master_slave_t
Master or slave mode.

typedef enum __sai_mono_stereo sai_mono_stereo_t
Mono or stereo audio format.

typedef enum __sai_data_order sai_data_order_t
SAI data order, MSB or LSB.

typedef enum __sai_clock_polarity sai_clock_polarity_t
SAI clock polarity, active high or low.

typedef enum __sai_sync_mode sai_sync_mode_t
Synchronous or asynchronous mode.

typedef enum __sai_bclk_source sai_bclk_source_t
Bit clock source.

typedef enum __sai_reset_type sai_reset_type_t
The reset type.

typedef enum __sai_fifo_packing sai_fifo_packing_t
The SAI packing mode The mode includes 8 bit and 16 bit packing.

typedef struct __sai_config sai_config_t
SAI user configuration structure.

typedef enum __sai_sample_rate sai_sample_rate_t
Audio sample rate.

typedef enum __sai_word_width sai_word_width_t
Audio word width.

```
typedef enum _sai_data_pin_state sai_data_pin_state_t
    sai data pin state definition

typedef enum _sai_fifo_combine sai_fifo_combine_t
    sai fifo combine mode definition

typedef enum _sai_transceiver_type sai_transceiver_type_t
    sai transceiver type

typedef enum _sai_frame_sync_len sai_frame_sync_len_t
    sai frame sync len

typedef struct _sai_transfer_format sai_transfer_format_t
    sai transfer format

typedef struct _sai_master_clock sai_master_clock_t
    master clock configurations

typedef struct _sai_fifo sai_fifo_t
    sai fifo configurations

typedef struct _sai_bit_clock sai_bit_clock_t
    sai bit clock configurations

typedef struct _sai_frame_sync sai_frame_sync_t
    sai frame sync configurations

typedef struct _sai_serial_data sai_serial_data_t
    sai serial data configurations

typedef struct _sai_transceiver sai_transceiver_t
    sai transceiver configurations

typedef struct _sai_transfer sai_transfer_t
    SAI transfer structure.

typedef struct _sai_handle sai_handle_t

typedef void (*sai_transfer_callback_t)(I2S_Type *base, sai_handle_t *handle, status_t status,
void *userData)
    SAI transfer callback prototype.
```

MCUX_SDK_SAI_ALLOW_NULL_FIFO_WATERMARK

Used to control whether SAI_RxSetFifoConfig()/SAI_TxSetFifoConfig() allows a NULL FIFO watermark.

If this macro is set to 0 then SAI_RxSetFifoConfig()/SAI_TxSetFifoConfig() will set the watermark to half of the FIFO's depth if passed a NULL watermark.

MCUX_SDK_SAI_DISABLE_IMPLICIT_CHAN_CONFIG

Disable implicit channel data configuration within SAI_TxSetConfig()/SAI_RxSetConfig().

Use this macro to control whether SAI_RxSetConfig()/SAI_TxSetConfig() will attempt to implicitly configure the channel data. By channel data we mean the startChannel, channelMask, endChannel, and channelNums fields from the sai_transciever_t structure. By default, SAI_TxSetConfig()/SAI_RxSetConfig() will attempt to compute these fields, which may not be desired in cases where the user wants to set them before the call to said functions.

SAI_XFER_QUEUE_SIZE

SAI transfer queue size, user can refine it according to use case.

FSL_SAI_HAS_FIFO_EXTEND_FEATURE

sai fifo feature

```
struct __sai_config
    #include <fsl_sai.h> SAI user configuration structure.
```

Public Members

```
sai_protocol_t protocol
    Audio bus protocol in SAI

sai_sync_mode_t syncMode
    SAI sync mode, control Tx/Rx clock sync

bool mclkOutputEnable
    Master clock output enable, true means master clock divider enabled

sai_bclk_source_t bclkSource
    Bit Clock source

sai_master_slave_t masterSlave
    Master or slave
```

```
struct __sai_transfer_format
    #include <fsl_sai.h> sai transfer format
```

Public Members

```
uint32_t sampleRate_Hz
    Sample rate of audio data

uint32_t bitWidth
    Data length of audio data, usually 8/16/24/32 bits

sai_mono_stereo_t stereo
    Mono or stereo

uint32_t masterClockHz
    Master clock frequency in Hz

uint8_t watermark
    Watermark value

uint8_t channel
    Transfer start channel

uint8_t channelMask
    enabled channel mask value, reference __sai_channel_mask

uint8_t endChannel
    end channel number

uint8_t channelNums
    Total enabled channel numbers

sai_protocol_t protocol
    Which audio protocol used

bool isFrameSyncCompact
    True means Frame sync length is configurable according to bitWidth, false means
    frame sync length is 64 times of bit clock.
```

```
struct __sai_master_clock
    #include <fsl_sai.h> master clock configurations
```


Public Members

bool mclkOutputEnable
master clock output enable

uint32_t mclkHz
target mclk frequency

uint32_t mclkSourceClkHz
mclk source frequency

struct _sai_fifo
#include <fsl_sai.h> sai fifo configurations

Public Members

bool fifoContinueOnError
fifo continues when error occur

sai_fifo_combine_t fifoCombine
fifo combine mode

sai_fifo_packing_t fifoPacking
fifo packing mode

uint8_t fifoWatermark
fifo watermark

struct _sai_bit_clock
#include <fsl_sai.h> sai bit clock configurations

Public Members

bool bclkInputDelay
bit clock actually used by the transmitter is delayed by the pad output delay, this has effect of decreasing the data input setup time, but increasing the data output valid time .

sai_clock_polarity_t bclkPolarity
bit clock polarity

sai_bclk_source_t bclkSource
bit Clock source

struct _sai_frame_sync
#include <fsl_sai.h> sai frame sync configurations

Public Members

uint8_t frameSyncWidth
frame sync width in number of bit clocks

bool frameSyncEarly
TRUE is frame sync assert one bit before the first bit of frame FALSE is frame sync assert with the first bit of the frame

bool frameSyncGenerateOnDemand
internal frame sync is generated when FIFO warning flag is clear

sai_clock_polarity_t frameSyncPolarity
frame sync polarity

struct *_sai_serial_data*
#include <fsl_sai.h> sai serial data configurations

Public Members

sai_data_pin_state_t dataMode
sai data pin state when slots masked or channel disabled

sai_data_order_t dataOrder
configure whether the LSB or MSB is transmitted first

uint8_t dataWord0Length
configure the number of bits in the first word in each frame

uint8_t dataWordNLength
configure the number of bits in the each word in each frame, except the first word

uint8_t dataWordLength
used to record the data length for dma transfer

uint8_t dataFirstBitShifted
Configure the bit index for the first bit transmitted for each word in the frame

uint8_t dataWordNum
configure the number of words in each frame

uint32_t dataMaskedWord
configure whether the transmit word is masked

struct *_sai_transceiver*
#include <fsl_sai.h> sai transceiver configurations

Public Members

sai_serial_data_t serialData
serial data configurations

sai_frame_sync_t frameSync
ws configurations

sai_bit_clock_t bitClock
bit clock configurations

sai_fifo_t fifo
fifo configurations

sai_master_slave_t masterSlave
transceiver is master or slave

sai_sync_mode_t syncMode
transceiver sync mode

uint8_t startChannel
Transfer start channel

uint8_t channelMask
enabled channel mask value, reference *_sai_channel_mask*

uint8_t endChannel
end channel number

uint8_t channelNums
Total enabled channel numbers

struct __sai_transfer
#include <fsl_sai.h> SAI transfer structure.

Public Members

uint8_t *data
Data start address to transfer.

size_t dataSize
Transfer size.

struct __sai_handle
#include <fsl_sai.h> SAI handle structure.

Public Members

I2S_Type *base
base address

uint32_t state
Transfer status

sai_transfer_callback_t callback
Callback function called at transfer event

void *userData
Callback parameter passed to callback function

uint8_t bitWidth
Bit width for transfer, 8/16/24/32 bits

uint8_t channel
Transfer start channel

uint8_t channelMask
enabled channel mask value, refernece *_sai_channel_mask*

uint8_t endChannel
end channel number

uint8_t channelNums
Total enabled channel numbers

sai_transfer_t saiQueue[(4U)]
Transfer queue storing queued transfer

size_t transferSize[(4U)]
Data bytes need to transfer

volatile uint8_t queueUser
Index for user to queue transfer

volatile uint8_t queueDriver
Index for driver to get the transfer data and size

uint8_t watermark
Watermark value

2.52 SAI EDMA Driver

```
void SAI_TransferTxCreateHandleEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                   sai_edma_callback_t callback, void *userData,  
                                   edma_handle_t *txDmaHandle)
```

Initializes the SAI eDMA handle.

This function initializes the SAI master DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- callback – Pointer to user callback function.
- userData – User parameter passed to the callback function.
- txDmaHandle – eDMA handle pointer, this handle shall be static allocated by users.

```
void SAI_TransferRxCreateHandleEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                   sai_edma_callback_t callback, void *userData,  
                                   edma_handle_t *rxDmaHandle)
```

Initializes the SAI Rx eDMA handle.

This function initializes the SAI slave DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- callback – Pointer to user callback function.
- userData – User parameter passed to the callback function.
- rxDmaHandle – eDMA handle pointer, this handle shall be static allocated by users.

```
void SAI_TransferSetInterleaveType(sai_edma_handle_t *handle, sai_edma_interleave_t  
                                  interleaveType)
```

Initializes the SAI interleave type.

This function initializes the SAI DMA handle member interleaveType, it shall be called only when application would like to use type kSAI_EDMAInterleavePerChannelBlock, since the default interleaveType is kSAI_EDMAInterleavePerChannelSample always

Parameters

- handle – SAI eDMA handle pointer.
- interleaveType – Multi channel interleave type.

```
void SAI_TransferTxSetConfigEDMA(I2S_Type *base, sai_edma_handle_t *handle,
                                sai_transceiver_t *saiConfig)
```

Configures the SAI Tx.

Note: SAI eDMA supports data transfer in a multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of sai_transceiver_t with the corresponding values of _sai_channel_mask enum, enable the FIFO Combine mode by assigning kSAI_FifoCombineModeEnabledOnWrite to the fifoCombine member of sai_fifo_combine_t which is a member of sai_transceiver_t. This is an example of multi-channel data transfer configuration step.

```
sai_transceiver_t config;
SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits, kSAI_Stereo, kSAI_Channel0Mask|kSAI_
↪Channel1Mask);
config.fifo.fifoCombine = kSAI_FifoCombineModeEnabledOnWrite;
SAI_TransferTxSetConfigEDMA(I2S0, &edmaHandle, &config);
```

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- saiConfig – sai configurations.

```
void SAI_TransferRxSetConfigEDMA(I2S_Type *base, sai_edma_handle_t *handle,
                                sai_transceiver_t *saiConfig)
```

Configures the SAI Rx.

Note: SAI eDMA supports data transfer in a multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of sai_transceiver_t with the corresponding values of _sai_channel_mask enum, enable the FIFO Combine mode by assigning kSAI_FifoCombineModeEnabledOnRead to the fifoCombine member of sai_fifo_combine_t which is a member of sai_transceiver_t. This is an example of multi-channel data transfer configuration step.

```
sai_transceiver_t config;
SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits, kSAI_Stereo, kSAI_Channel0Mask|kSAI_
↪Channel1Mask);
config.fifo.fifoCombine = kSAI_FifoCombineModeEnabledOnRead;
SAI_TransferRxSetConfigEDMA(I2S0, &edmaHandle, &config);
```

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- saiConfig – sai configurations.

```
status_t SAI_TransferSendEDMA(I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t
                                *xfer)
```

Performs a non-blocking SAI transfer using DMA.

This function support multi channel transfer,

- a. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
- b. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Note: This interface returns immediately after the transfer initiates. Call `SAI_GetTransferStatus` to poll the transfer status and check whether the SAI transfer is finished.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- xfer – Pointer to the DMA transfer structure.

Return values

- `kStatus_Success` – Start a SAI eDMA send successfully.
- `kStatus_InvalidArgument` – The input argument is invalid.
- `kStatus_TxBusy` – SAI is busy sending data.

status_t `SAI_TransferReceiveEDMA(I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t *xfer)`

Performs a non-blocking SAI receive using eDMA.

This function support multi channel transfer,

- a. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
- b. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Note: This interface returns immediately after the transfer initiates. Call the `SAI_GetReceiveRemainingBytes` to poll the transfer status and check whether the SAI transfer is finished.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- `kStatus_Success` – Start a SAI eDMA receive successfully.
- `kStatus_InvalidArgument` – The input argument is invalid.
- `kStatus_RxBusy` – SAI is busy receiving data.

status_t `SAI_TransferSendLoopEDMA(I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t *xfer, uint32_t loopTransferCount)`

Performs a non-blocking SAI loop transfer using eDMA.

Once the loop transfer start, application can use function `SAI_TransferAbortSendEDMA` to stop the loop transfer.

Note: This function support loop transfer only,such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in `sai_edma_handle_t`, so application could redefine the `SAI_XFER_QUEUE_SIZE` to determine the proper TCD pool size. This function support one sai channel only.

Parameters

- `base` – SAI base pointer.
- `handle` – SAI eDMA handle pointer.
- `xfer` – Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (`loopTransferCount`).
- `loopTransferCount` – the counts of `xfer` array.

Return values

- `kStatus_Success` – Start a SAI eDMA send successfully.
- `kStatus_InvalidArgument` – The input argument is invalid.

`status_t` `SAI_TransferReceiveLoopEDMA(I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t *xfer, uint32_t loopTransferCount)`

Performs a non-blocking SAI loop transfer using eDMA.

Once the loop transfer start, application can use function `SAI_TransferAbortReceiveEDMA` to stop the loop transfer.

Note: This function support loop transfer only,such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in `sai_edma_handle_t`, so application could redefine the `SAI_XFER_QUEUE_SIZE` to determine the proper TCD pool size. This function support one sai channel only.

Parameters

- `base` – SAI base pointer.
- `handle` – SAI eDMA handle pointer.
- `xfer` – Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (`loopTransferCount`).
- `loopTransferCount` – the counts of `xfer` array.

Return values

- `kStatus_Success` – Start a SAI eDMA receive successfully.
- `kStatus_InvalidArgument` – The input argument is invalid.

`void` `SAI_TransferTerminateSendEDMA(I2S_Type *base, sai_edma_handle_t *handle)`

Terminate all SAI send.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call `SAI_TransferAbortSendEDMA`.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferTerminateReceiveEDMA(I2S_Type *base, sai_edma_handle_t *handle)

Terminate all SAI receive.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceiveEDMA.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferAbortSendEDMA(I2S_Type *base, sai_edma_handle_t *handle)

Aborts a SAI transfer using eDMA.

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateSendEDMA.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferAbortReceiveEDMA(I2S_Type *base, sai_edma_handle_t *handle)

Aborts a SAI receive using eDMA.

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateReceiveEDMA.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.

status_t SAI_TransferGetSendCountEDMA(I2S_Type *base, sai_edma_handle_t *handle, size_t *count)

Gets byte count sent by SAI.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- count – Bytes count sent by SAI.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is no non-blocking transaction in progress.

status_t SAI_TransferGetReceiveCountEDMA(I2S_Type *base, sai_edma_handle_t *handle, size_t *count)

Gets byte count received by SAI.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.

- `count` – Bytes count received by SAI.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is no non-blocking transaction in progress.

`uint32_t SAI_TransferGetValidTransferSlotsEDMA(I2S_Type *base, sai_edma_handle_t *handle)`

Gets valid transfer slot.

This function can be used to query the valid transfer request slot that the application can submit. It should be called in the critical section, that means the application could call it in the corresponding callback function or disable IRQ before calling it in the application, otherwise, the returned value may not correct.

Parameters

- `base` – SAI base pointer
- `handle` – SAI eDMA handle pointer.

Return values

`valid` – slot count that application submit.

`FSL_SAI_EDMA_DRIVER_VERSION`

Version 2.7.3

`enum _sai_edma_interleave`

sai interleave type

Values:

enumerator `kSAI_EDMAInterleavePerChannelSample`

enumerator `kSAI_EDMAInterleavePerChannelBlock`

`typedef struct sai_edma_handle sai_edma_handle_t`

`typedef void (*sai_edma_callback_t)(I2S_Type *base, sai_edma_handle_t *handle, status_t status, void *userData)`

SAI eDMA transfer callback function for finish and error.

`typedef enum _sai_edma_interleave sai_edma_interleave_t`

sai interleave type

`MCUX_SDK_SAI_EDMA_RX_ENABLE_INTERNAL`

the SAI enable position When calling `SAI_TransferReceiveEDMA`

`MCUX_SDK_SAI_EDMA_TX_ENABLE_INTERNAL`

the SAI enable position When calling `SAI_TransferSendEDMA`

`struct sai_edma_handle`

`#include <fsl_sai_edma.h>` SAI DMA transfer handle, users should not touch the content of the handle.

Public Members

`edma_handle_t *dmaHandle`

DMA handler for SAI send

`uint8_t nbytes`

eDMA minor byte transfer count initially configured.

`uint8_t bytesPerFrame`
Bytes in a frame

`uint8_t channelMask`
Enabled channel mask value, reference `_sai_channel_mask`

`uint8_t channelNums`
total enabled channel nums

`uint8_t channel`
Which data channel

`uint8_t count`
The transfer data count in a DMA request

`uint32_t state`
Internal state for SAI eDMA transfer

`sai_edma_callback_t callback`
Callback for users while transfer finish or error occurs

`void *userData`
User callback parameter

`uint8_t tcd[((4U) + 1U) * sizeof(edma_tcd_t)]`
TCD pool for eDMA transfer.

`sai_transfer_t saiQueue[(4U)]`
Transfer queue storing queued transfer.

`size_t transferSize[(4U)]`
Data bytes need to transfer

`sai_edma_interleave_t interleaveType`
Transfer interleave type

`volatile uint8_t queueUser`
Index for user to queue transfer.

`volatile uint8_t queueDriver`
Index for driver to get the transfer data and size

2.53 SNVS: Secure Non-Volatile Storage

2.54 Secure Non-Volatile Storage High-Power

`void SNVS_HP_Init(SNVS_Type *base)`
Initialize the SNVS.

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

Parameters

- `base` – SNVS peripheral base address

void SNVS_HP_Deinit(SNVS_Type *base)

Deinitialize the SNVS.

Parameters

- base – SNVS peripheral base address

void SNVS_HP_RTC_Init(SNVS_Type *base, const *snvs_hp_rtc_config_t* *config)

Ungates the SNVS clock and configures the peripheral for basic operation.

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

Parameters

- base – SNVS peripheral base address
- config – Pointer to the user's SNVS configuration structure.

void SNVS_HP_RTC_Deinit(SNVS_Type *base)

Stops the RTC and SRTC timers.

Parameters

- base – SNVS peripheral base address

void SNVS_HP_RTC_GetDefaultConfig(*snvs_hp_rtc_config_t* *config)

Fills in the SNVS config struct with the default settings.

The default values are as follows.

```
config->rtccalenable = false;
config->rtccalvalue = 0U;
config->PIFreq = 0U;
```

Parameters

- config – Pointer to the user's SNVS configuration structure.

status_t SNVS_HP_RTC_SetDatetime(SNVS_Type *base, const *snvs_hp_rtc_datetime_t* *datetime)

Sets the SNVS RTC date and time according to the given time structure.

Parameters

- base – SNVS peripheral base address
- datetime – Pointer to the structure where the date and time details are stored.

Returns

kStatus_Success: Success in setting the time and starting the SNVS RTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

void SNVS_HP_RTC_GetDatetime(SNVS_Type *base, *snvs_hp_rtc_datetime_t* *datetime)

Gets the SNVS RTC time and stores it in the given time structure.

Parameters

- base – SNVS peripheral base address
- datetime – Pointer to the structure where the date and time details are stored.

```
status_t SNVS_HP_RTC_SetAlarm(SNVS_Type *base, const snvs_hp_rtc_datetime_t
                             *alarmTime)
```

Sets the SNVS RTC alarm time.

The function sets the RTC alarm. It also 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 – SNVS peripheral base address
- alarmTime – Pointer to the structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the SNVS RTC alarm

kStatus_InvalidArgument: Error because the alarm datetime format is incorrect

kStatus_Fail: Error because the alarm time has already passed

```
void SNVS_HP_RTC_GetAlarm(SNVS_Type *base, snvs_hp_rtc_datetime_t *datetime)
```

Returns the SNVS RTC alarm time.

Parameters

- base – SNVS peripheral base address
- datetime – Pointer to the structure where the alarm date and time details are stored.

```
void SNVS_HP_RTC_TimeSynchronize(SNVS_Type *base)
```

The function synchronizes RTC counter value with SRTC.

Parameters

- base – SNVS peripheral base address

```
static inline void SNVS_HP_RTC_EnableInterrupts(SNVS_Type *base, uint32_t mask)
```

Enables the selected SNVS interrupts.

Parameters

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

```
static inline void SNVS_HP_RTC_DisableInterrupts(SNVS_Type *base, uint32_t mask)
```

Disables the selected SNVS interrupts.

Parameters

- base – SNVS peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration :: `_snvs_hp_interrupts_t`

```
uint32_t SNVS_HP_RTC_GetEnabledInterrupts(SNVS_Type *base)
```

Gets the enabled SNVS interrupts.

Parameters

- base – SNVS peripheral base address

Returns

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

:: `_snvs_hp_interrupts_t`

uint32_t SNVS_HP_RTC_GetStatusFlags(SNVS_Type *base)

Gets the SNVS status flags.

Parameters

- base – SNVS peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration ::
_snvs_hp_status_flags_t

static inline void SNVS_HP_RTC_ClearStatusFlags(SNVS_Type *base, uint32_t mask)

Clears the SNVS status flags.

Parameters

- base – SNVS peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration :: _snvs_hp_status_flags_t

static inline void SNVS_HP_RTC_StartTimer(SNVS_Type *base)

Starts the SNVS RTC time counter.

Parameters

- base – SNVS peripheral base address

static inline void SNVS_HP_RTC_StopTimer(SNVS_Type *base)

Stops the SNVS RTC time counter.

Parameters

- base – SNVS peripheral base address

static inline void SNVS_HP_EnableHighAssuranceCounter(SNVS_Type *base, bool enable)

Enable or disable the High Assurance Counter (HAC)

Parameters

- base – SNVS peripheral base address
- enable – Pass true to enable, false to disable.

static inline void SNVS_HP_StartHighAssuranceCounter(SNVS_Type *base, bool start)

Start or stop the High Assurance Counter (HAC)

Parameters

- base – SNVS peripheral base address
- start – Pass true to start, false to stop.

static inline void SNVS_HP_SetHighAssuranceCounterInitialValue(SNVS_Type *base, uint32_t value)

Set the High Assurance Counter (HAC) initialize value.

Parameters

- base – SNVS peripheral base address
- value – The initial value to set.

static inline void SNVS_HP_LoadHighAssuranceCounter(SNVS_Type *base)

Load the High Assurance Counter (HAC)

This function loads the HAC initialize value to counter register.

Parameters

- base – SNVS peripheral base address

```
static inline uint32_t SNVS_HP_GetHighAssuranceCounter(SNVS_Type *base)
```

Get the current High Assurance Counter (HAC) value.

Parameters

- base – SNVS peripheral base address

Returns

HAC current value.

```
static inline void SNVS_HP_ClearHighAssuranceCounter(SNVS_Type *base)
```

Clear the High Assurance Counter (HAC)

This function can be called in a functional or soft fail state. When the HAC is enabled:

- If the HAC is cleared in the soft fail state, the SSM transitions to the hard fail state immediately;
- If the HAC is cleared in functional state, the SSM will transition to hard fail immediately after transitioning to soft fail.

Parameters

- base – SNVS peripheral base address

```
static inline void SNVS_HP_LockHighAssuranceCounter(SNVS_Type *base)
```

Lock the High Assurance Counter (HAC)

Once locked, the HAC initialize value could not be changed, the HAC enable status could not be changed. This could only be unlocked by system reset.

Parameters

- base – SNVS peripheral base address

```
FSL_SNVS_HP_DRIVER_VERSION
```

Version 2.3.2

```
enum _snvs_hp_interrupts
```

List of SNVS interrupts.

Values:

```
enumerator kSNVS_RTC_AlarmInterrupt
```

RTC time alarm

```
enumerator kSNVS_RTC_PeriodicInterrupt
```

RTC periodic interrupt

```
enum _snvs_hp_status_flags
```

List of SNVS flags.

Values:

```
enumerator kSNVS_RTC_AlarmInterruptFlag
```

RTC time alarm flag

```
enumerator kSNVS_RTC_PeriodicInterruptFlag
```

RTC periodic interrupt flag

```
enumerator kSNVS_ZMK_ZeroFlag
```

The ZMK is zero

```
enumerator kSNVS_OTPMK_ZeroFlag
```

The OTPMK is zero

enum _snvs_hp_sv_status_flags

List of SNVS security violation flags.

Values:

enumerator kSNVS_LP_ViolationFlag

Low Power section Security Violation

enumerator kSNVS_ZMK_EccFailFlag

Zeroizable Master Key Error Correcting Code Check Failure

enumerator kSNVS_LP_SoftwareViolationFlag

LP Software Security Violation

enumerator kSNVS_FatalSoftwareViolationFlag

Software Fatal Security Violation

enumerator kSNVS_SoftwareViolationFlag

Software Security Violation

enumerator kSNVS_Violation0Flag

Security Violation 0

enumerator kSNVS_Violation1Flag

Security Violation 1

enumerator kSNVS_Violation2Flag

Security Violation 2

enumerator kSNVS_Violation4Flag

Security Violation 4

enumerator kSNVS_Violation5Flag

Security Violation 5

enum _snvs_hp_ssm_state

List of SNVS Security State Machine State.

Values:

enumerator kSNVS_SSMInit

Init

enumerator kSNVS_SSMHardFail

Hard Fail

enumerator kSNVS_SSMSoftFail

Soft Fail

enumerator kSNVS_SSMInitInter

Init Intermediate (transition state between Init and Check)

enumerator kSNVS_SSMCheck

Check

enumerator kSNVS_SSMNonSecure

Non-Secure

enumerator kSNVS_SSMTrusted

Trusted

enumerator kSNVS_SSMSecure

Secure

```
typedef enum _snvs_hp_interrupts snvs_hp_interrupts_t
```

List of SNVS interrupts.

```
typedef enum _snvs_hp_status_flags snvs_hp_status_flags_t
```

List of SNVS flags.

```
typedef enum _snvs_hp_sv_status_flags snvs_hp_sv_status_flags_t
```

List of SNVS security violation flags.

```
typedef struct _snvs_hp_rtc_datetime snvs_hp_rtc_datetime_t
```

Structure is used to hold the date and time.

```
typedef struct _snvs_hp_rtc_config snvs_hp_rtc_config_t
```

SNVS config structure.

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

The config struct can be made const so it resides in flash

```
typedef enum _snvs_hp_ssm_state snvs_hp_ssm_state_t
```

List of SNVS Security State Machine State.

```
static inline void SNVS_HP_EnableMasterKeySelection(SNVS_Type *base, bool enable)
```

Enable or disable master key selection.

Parameters

- base – SNVS peripheral base address
- enable – Pass true to enable, false to disable.

```
static inline void SNVS_HP_ProgramZeroizableMasterKey(SNVS_Type *base)
```

Trigger to program Zeroizable Master Key.

Parameters

- base – SNVS peripheral base address

```
static inline void SNVS_HP_ChangeSSMState(SNVS_Type *base)
```

Trigger SSM State Transition.

Trigger state transition of the system security monitor (SSM). It results only the following transitions of the SSM:

- Check State -> Non-Secure (when Non-Secure Boot and not in Fab Configuration)
- Check State -> Trusted (when Secure Boot or in Fab Configuration)
- Trusted State -> Secure
- Secure State -> Trusted
- Soft Fail -> Non-Secure

Parameters

- base – SNVS peripheral base address

```
static inline void SNVS_HP_SetSoftwareFatalSecurityViolation(SNVS_Type *base)
```

Trigger Software Fatal Security Violation.

The result SSM state transition is:

- Check State -> Soft Fail
- Non-Secure State -> Soft Fail

- Trusted State -> Soft Fail
- Secure State -> Soft Fail

Parameters

- base – SNVS peripheral base address

static inline void SNVS_HP_SetSoftwareSecurityViolation(SNVS_Type *base)

Trigger Software Security Violation.

The result SSM state transition is:

- Check -> Non-Secure
- Trusted -> Soft Fail
- Secure -> Soft Fail

Parameters

- base – SNVS peripheral base address

static inline *snvs_hp_ssm_state_t* SNVS_HP_GetSSMState(SNVS_Type *base)

Get current SSM State.

Parameters

- base – SNVS peripheral base address

Returns

Current SSM state

static inline void SNVS_HP_ResetLP(SNVS_Type *base)

Reset the SNVS LP section.

Reset the LP section except SRTC and Time alarm.

Parameters

- base – SNVS peripheral base address

static inline uint32_t SNVS_HP_GetStatusFlags(SNVS_Type *base)

Get the SNVS HP status flags.

The flags are returned as the OR'ed value of the enumeration :: *_snvs_hp_status_flags_t*.

Parameters

- base – SNVS peripheral base address

Returns

The OR'ed value of status flags.

static inline void SNVS_HP_ClearStatusFlags(SNVS_Type *base, uint32_t mask)

Clear the SNVS HP status flags.

The flags to clear are passed in as the OR'ed value of the enumeration :: *_snvs_hp_status_flags_t*. Only these flags could be cleared using this API.

- kSNVS_RTC_PeriodicInterruptFlag
- kSNVS_RTC_AlarmInterruptFlag

Parameters

- base – SNVS peripheral base address
- mask – OR'ed value of the flags to clear.

```
static inline uint32_t SNVS_HP_GetSecurityViolationStatusFlags(SNVS_Type *base)
```

Get the SNVS HP security violation status flags.

The flags are returned as the OR'ed value of the enumeration :: `_snvs_hp_sv_status_flags_t`.

Parameters

- `base` – SNVS peripheral base address

Returns

The OR'ed value of security violation status flags.

```
static inline void SNVS_HP_ClearSecurityViolationStatusFlags(SNVS_Type *base, uint32_t mask)
```

Clear the SNVS HP security violation status flags.

The flags to clear are passed in as the OR'ed value of the enumeration :: `_snvs_hp_sv_status_flags_t`. Only these flags could be cleared using this API.

- `kSNVS_ZMK_EccFailFlag`
- `kSNVS_Violation0Flag`
- `kSNVS_Violation1Flag`
- `kSNVS_Violation2Flag`
- `kSNVS_Violation3Flag`
- `kSNVS_Violation4Flag`
- `kSNVS_Violation5Flag`

Parameters

- `base` – SNVS peripheral base address
- `mask` – OR'ed value of the flags to clear.

```
SNVS_HPSVSR_SV0_MASK
```

```
SNVS_HPSVSR_SV1_MASK
```

```
SNVS_HPSVSR_SV2_MASK
```

```
SNVS_HPSVSR_SV4_MASK
```

```
SNVS_HPSVSR_SV5_MASK
```

```
SNVS_MAKE_HP_SV_FLAG(x)
```

Macro to make security violation flag.

Macro help to make security violation flag `kSNVS_Violation0Flag` to `kSNVS_Violation5Flag`. For example, `SNVS_MAKE_HP_SV_FLAG(0)` is `kSNVS_Violation0Flag`.

```
struct _snvs_hp_rtc_datetime
```

#include <fsl_snvs_hp.h> Structure is used to hold the date and time.

Public Members

```
uint16_t year
```

Range from 1970 to 2099.

```
uint8_t month
```

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

struct __snvs_hp_rtc_config

#include <fsl_snvs_hp.h> SNVS config structure.

This structure holds the configuration settings for the SNVS peripheral. To initialize this structure to reasonable defaults, call the SNVS_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 rtcCalEnable

true: RTC calibration mechanism is enabled; false: No calibration is used

uint32_t rtcCalValue

Defines signed calibration value for nonsecure RTC; This is a 5-bit 2's complement value, range from -16 to +15

uint32_t periodicInterruptFreq

Defines frequency of the periodic interrupt; Range from 0 to 15

2.55 Secure Non-Volatile Storage Low-Power

void SNVS_LP_Init(SNVS_Type *base)

Un gates the SNVS clock and configures the peripheral for basic operation.

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

Parameters

- base – SNVS peripheral base address

void SNVS_LP_Deinit(SNVS_Type *base)

Deinit the SNVS LP section.

Parameters

- base – SNVS peripheral base address

status_t SNVS_LP_SRTC_SetDatetime(SNVS_Type *base, const snvs_lp_srtc_datetime_t *datetime)

Sets the SNVS SRTC date and time according to the given time structure.

Parameters

- base – SNVS peripheral base address

- `datetime` – Pointer to the structure where the date and time details are stored.

Returns

`kStatus_Success`: Success in setting the time and starting the SNVS SRTC
`kStatus_InvalidArgument`: Error because the datetime format is incorrect

```
void SNVS_LP_SRTC_GetDatetime(SNVS_Type *base, snvs_lp_srtc_datetime_t *datetime)
```

Gets the SNVS SRTC time and stores it in the given time structure.

Parameters

- `base` – SNVS peripheral base address
- `datetime` – Pointer to the structure where the date and time details are stored.

```
status_t SNVS_LP_SRTC_SetAlarm(SNVS_Type *base, const snvs_lp_srtc_datetime_t *alarmTime)
```

Sets the SNVS SRTC alarm time.

The function sets the SRTC alarm. It also 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. Please note, that SRTC alarm has limited resolution because only 32 most significant bits of SRTC counter are compared to SRTC Alarm register. If the alarm time is beyond SRTC resolution, the function does not set the alarm and returns an error.

Parameters

- `base` – SNVS peripheral base address
- `alarmTime` – Pointer to the structure where the alarm time is stored.

Returns

`kStatus_Success`: success in setting the SNVS SRTC alarm
`kStatus_InvalidArgument`: Error because the alarm datetime format is incorrect
`kStatus_Fail`: Error because the alarm time has already passed or is beyond resolution

```
void SNVS_LP_SRTC_GetAlarm(SNVS_Type *base, snvs_lp_srtc_datetime_t *datetime)
```

Returns the SNVS SRTC alarm time.

Parameters

- `base` – SNVS peripheral base address
- `datetime` – Pointer to the structure where the alarm date and time details are stored.

```
static inline void SNVS_LP_SRTC_EnableInterrupts(SNVS_Type *base, uint32_t mask)
```

Enables the selected SNVS interrupts.

Parameters

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

```
static inline void SNVS_LP_SRTC_DisableInterrupts(SNVS_Type *base, uint32_t mask)
```

Disables the selected SNVS interrupts.

Parameters

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

uint32_t SNVS_LP_SRTC_GetEnabledInterrupts(SNVS_Type *base)

Gets the enabled SNVS interrupts.

Parameters

- base – SNVS peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration :: *_snvs_lp_srtc_interrupts*

uint32_t SNVS_LP_SRTC_GetStatusFlags(SNVS_Type *base)

Gets the SNVS status flags.

Parameters

- base – SNVS peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration :: *_snvs_lp_srtc_status_flags*

static inline void SNVS_LP_SRTC_ClearStatusFlags(SNVS_Type *base, uint32_t mask)

Clears the SNVS status flags.

Parameters

- base – SNVS peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration :: *_snvs_lp_srtc_status_flags*

static inline void SNVS_LP_SRTC_StartTimer(SNVS_Type *base)

Starts the SNVS SRTC time counter.

Parameters

- base – SNVS peripheral base address

static inline void SNVS_LP_SRTC_StopTimer(SNVS_Type *base)

Stops the SNVS SRTC time counter.

Parameters

- base – SNVS peripheral base address

void SNVS_LP_EnablePassiveTamper(SNVS_Type *base, *snvs_lp_external_tamper_t* pin, *snvs_lp_passive_tamper_t* config)

Enables the specified SNVS external tamper.

Parameters

- base – SNVS peripheral base address
- pin – SNVS external tamper pin
- config – Configuration structure of external passive tamper

status_t SNVS_LP_EnableTxActiveTamper(SNVS_Type *base, *snvs_lp_active_tx_tamper_t* pin, *tamper_active_tx_config_t* config)

Enable active tamper tx external pad.

Parameters

- base – SNVS peripheral base address
- pin – SNVS active tamper pin
- config – Configuration structure of external active tamper

status_t SNVS_LP_EnableRxActiveTamper(SNVS_Type *base, *snvs_lp_external_tamper_t* rx, *tamper_active_rx_config_t* config)

Enable active tamper rx external pad.

Parameters

- base – SNVS peripheral base address
- rx – SNVS external RX tamper pin
- config – SNVS RX tamper config structure

status_t SNVS_LP_SetVoltageTamper(SNVS_Type *base, bool enable)

Sets voltage tamper detect.

Parameters

- base – SNVS peripheral base address
- enable – True if enable false if disable

status_t SNVS_LP_SetTemperatureTamper(SNVS_Type *base, bool enable)

Sets temperature tamper detect.

Parameters

- base – SNVS peripheral base address
- enable – True if enable false if disable

status_t SNVS_LP_SetClockTamper(SNVS_Type *base, bool enable)

Sets clock tamper detect.

Parameters

- base – SNVS peripheral base address
- enable – True if enable false if disable

snvs_lp_external_tamper_status_t SNVS_LP_CheckVoltageTamper(SNVS_Type *base)

brief Check voltage tamper

param base SNVS peripheral base address

snvs_lp_external_tamper_status_t SNVS_LP_CheckTemperatureTamper(SNVS_Type *base)

Check temperature tamper.

Parameters

- base – SNVS peripheral base address

snvs_lp_external_tamper_status_t SNVS_LP_CheckClockTamper(SNVS_Type *base)

brief Check clock tamper

param base SNVS peripheral base address

void SNVS_LP_TamperPinTx_GetDefaultConfig(*tamper_active_tx_config_t* *config)

Fills in the SNVS tamper pin config struct with the default settings.

The default values are as follows. code config->clock = kSNVS_ActiveTamper16HZ; config->seed = 0U; config->polynomial = 0U; endcode

Parameters

- config – Pointer to the user's SNVS configuration structure.

`void SNVS_LP_TamperPinRx_GetDefaultConfig(tamper_active_rx_config_t *config)`

brief Fills in the SNVS tamper pin config struct with the default settings.

The default values are as follows. `config->filterenable = 0U`; `config->filter = 0U`; `config->tx = kSNVS_ActiveTamper1`; endcode param config Pointer to the user's SNVS configuration structure.

`void SNVS_LP_PassiveTamperPin_GetDefaultConfig(snvs_lp_passive_tamper_t *config)`

Fills in the SNVS tamper pin config struct with the default settings.

The default values are as follows. `config->polarity = 0U`; `config->filterenable = 0U`; if available on SoC `config->filter = 0U`; if available on SoC endcode

Parameters

- `config` – Pointer to the user's SNVS configuration structure.

`void SNVS_LP_DisableExternalTamper(SNVS_Type *base, snvs_lp_external_tamper_t pin)`

Disables the specified SNVS external tamper.

Parameters

- `base` – SNVS peripheral base address
- `pin` – SNVS external tamper pin

`void SNVS_LP_DisableAllExternalTamper(SNVS_Type *base)`

Disable all external tamper.

Parameters

- `base` – SNVS peripheral base address

`snvs_lp_external_tamper_status_t` SNVS_LP_GetExternalTamperStatus(SNVS_Type *base, *snvs_lp_external_tamper_t* pin)

Returns status of the specified external tamper.

Parameters

- `base` – SNVS peripheral base address
- `pin` – SNVS external tamper pin

Returns

The status flag. This is the enumeration :: `_snvs_lp_external_tamper_status`

`void SNVS_LP_ClearExternalTamperStatus(SNVS_Type *base, snvs_lp_external_tamper_t pin)`

Clears status of the specified external tamper.

Parameters

- `base` – SNVS peripheral base address
- `pin` – SNVS external tamper pin

`void SNVS_LP_ClearAllExternalTamperStatus(SNVS_Type *base)`

Clears status of the all external tamper.

Parameters

- `base` – SNVS peripheral base address

`static inline void SNVS_LP_EnableMonotonicCounter(SNVS_Type *base, bool enable)`

Enable or disable the Monotonic Counter.

Parameters

- `base` – SNVS peripheral base address

- enable – Pass true to enable, false to disable.

uint64_t SNVS_LP_GetMonotonicCounter(SNVS_Type *base)

Get the current Monotonic Counter.

Parameters

- base – SNVS peripheral base address

Returns

Current Monotonic Counter value.

static inline void SNVS_LP_IncreaseMonotonicCounter(SNVS_Type *base)

Increase the Monotonic Counter.

Increase the Monotonic Counter by 1.

Parameters

- base – SNVS peripheral base address

void SNVS_LP_WriteZeroizableMasterKey(SNVS_Type *base, uint32_t ZMKey[8U])

Write Zeroizable Master Key (ZMK) to the SNVS registers.

Parameters

- base – SNVS peripheral base address
- ZMKey – The ZMK write to the SNVS register.

static inline void SNVS_LP_SetZeroizableMasterKeyValid(SNVS_Type *base, bool valid)

Set Zeroizable Master Key valid.

This API could only be called when using software programming mode. After writing ZMK using SNVS_LP_WriteZeroizableMasterKey, call this API to make the ZMK valid.

Parameters

- base – SNVS peripheral base address
- valid – Pass true to set valid, false to set invalid.

static inline bool SNVS_LP_GetZeroizableMasterKeyValid(SNVS_Type *base)

Get Zeroizable Master Key valid status.

In hardware programming mode, call this API to check whether the ZMK is valid.

Parameters

- base – SNVS peripheral base address

Returns

true if valid, false if invalid.

static inline void SNVS_LP_SetZeroizableMasterKeyProgramMode(SNVS_Type *base,
snvs_lp_zmk_program_mode_t mode)

Set Zeroizable Master Key programming mode.

Parameters

- base – SNVS peripheral base address
- mode – ZMK programming mode.

static inline void SNVS_LP_EnableZeroizableMasterKeyECC(SNVS_Type *base, bool enable)

Enable or disable Zeroizable Master Key ECC.

Parameters

- base – SNVS peripheral base address

- enable – Pass true to enable, false to disable.

```
static inline void SNVS_LP_SetMasterKeyMode(SNVS_Type *base, snvs_lp_master_key_mode_t mode)
```

Set SNVS Master Key mode.

Note: When kSNVS_ZMK or kSNVS_CMK used, the SNVS_HP must be configured to enable the master key selection.

Parameters

- base – SNVS peripheral base address
- mode – Master Key mode.

FSL_SNVS_LP_DRIVER_VERSION

Version 2.4.6

enum _snvs_lp_srtc_interrupts

List of SNVS_LP interrupts.

Values:

enumerator kSNVS_SRTC_AlarmInterrupt
SRTC time alarm.

enum _snvs_lp_srtc_status_flags

List of SNVS_LP flags.

Values:

enumerator kSNVS_SRTC_AlarmInterruptFlag
SRTC time alarm flag

enum _snvs_lp_external_tamper

List of SNVS_LP external tamperers.

Values:

enumerator kSNVS_ExternalTamper1

enum _snvs_lp_active_tamper

List of SNVS_LP active tamperers.

Values:

enumerator kSNVS_ActiveTamper1

enumerator kSNVS_ActiveTamper2

enumerator kSNVS_ActiveTamper3

enumerator kSNVS_ActiveTamper4

enumerator kSNVS_ActiveTamper5

enum _snvs_lp_active_clock

List of SNVS_LP external tamperers.

Values:

enumerator kSNVS_ActiveTamper16HZ

enumerator kSNVS_ActiveTamper8HZ

enumerator kSNVS__ActiveTamper4HZ

enumerator kSNVS__ActiveTamper2HZ

enum _snvs_lp_external_tamper_status

List of SNVS_LP external tampers status.

Values:

enumerator kSNVS__TamperNotDetected

enumerator kSNVS__TamperDetected

enum _snvs_lp_external_tamper_polarity

SNVS_LP external tamper polarity.

Values:

enumerator kSNVS__ExternalTamperActiveLow

enumerator kSNVS__ExternalTamperActiveHigh

enum _snvs_lp_zmk_program_mode

SNVS_LP Zeroizable Master Key programming mode.

Values:

enumerator kSNVS__ZMKSoftwareProgram

Software programming mode.

enumerator kSNVS__ZMKHardwareProgram

Hardware programming mode.

enum _snvs_lp_master_key_mode

SNVS_LP Master Key mode.

Values:

enumerator kSNVS__OTPMK

One Time Programmable Master Key.

enumerator kSNVS__ZMK

Zeroizable Master Key.

enumerator kSNVS__CMK

Combined Master Key, it is XOR of OPTMK and ZMK.

typedef enum _snvs_lp_srtc_interrupts snvs_lp_srtc_interrupts_t

List of SNVS_LP interrupts.

typedef enum _snvs_lp_srtc_status_flags snvs_lp_srtc_status_flags_t

List of SNVS_LP flags.

typedef enum _snvs_lp_external_tamper snvs_lp_external_tamper_t

List of SNVS_LP external tampers.

typedef enum _snvs_lp_active_tamper snvs_lp_active_tx_tamper_t

List of SNVS_LP active tampers.

typedef enum _snvs_lp_active_clock snvs_lp_active_clock_t

List of SNVS_LP external tampers.

typedef enum _snvs_lp_external_tamper_status snvs_lp_external_tamper_status_t

List of SNVS_LP external tampers status.

```
typedef enum _snvs_lp_external_tamper_polarity snvs_lp_external_tamper_polarity_t
    SNVS_LP external tamper polarity.
```

```
typedef struct _snvs_lp_srtc_datetime snvs_lp_srtc_datetime_t
    Structure is used to hold the date and time.
```

```
typedef struct _snvs_lp_srtc_config snvs_lp_srtc_config_t
    SNVS_LP config structure.
```

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

The config struct can be made const so it resides in flash

```
typedef enum _snvs_lp_zmk_program_mode snvs_lp_zmk_program_mode_t
    SNVS_LP Zeroizable Master Key programming mode.
```

```
typedef enum _snvs_lp_master_key_mode snvs_lp_master_key_mode_t
    SNVS_LP Master Key mode.
```

```
void SNVS_LP_SRTC_Init(SNVS_Type *base, const snvs_lp_srtc_config_t *config)
    Ungates the SNVS clock and configures the peripheral for basic operation.
```

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

Parameters

- base – SNVS peripheral base address
- config – Pointer to the user's SNVS configuration structure.

```
void SNVS_LP_SRTC_Deinit(SNVS_Type *base)
    Stops the SRTC timer.
```

Parameters

- base – SNVS peripheral base address

```
void SNVS_LP_SRTC_GetDefaultConfig(snvs_lp_srtc_config_t *config)
    Fills in the SNVS_LP config struct with the default settings.
```

The default values are as follows.

```
config->srtccalenable = false;
config->srtccalvalue = 0U;
```

Parameters

- config – Pointer to the user's SNVS configuration structure.

```
SNVS_ZMK_REG_COUNT
    Define of SNVS_LP Zeroizable Master Key registers.
```

```
SNVS_LP_MAX_TAMPER
    Define of SNVS_LP Max possible tamper.
```

```
struct tamper_active_tx_config_t
    #include <fsl_snvs_lp.h> Structure is used to configure SNVS LP active TX tamper pins.
```

```
struct tamper_active_rx_config_t
    #include <fsl_snvs_lp.h> Structure is used to configure SNVS LP active RX tamper pins.
```

struct snvs_lp_passive_tamper_t

#include <fsl_snvs_lp.h> Structure is used to configure SNVS LP passive tamper pins.

struct __snvs_lp_srtc_datetime

#include <fsl_snvs_lp.h> Structure is used to hold the date and time.

Public Members

uint16_t year

Range from 1970 to 2099.

uint8_t month

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

struct __snvs_lp_srtc_config

#include <fsl_snvs_lp.h> SNVS_LP config structure.

This structure holds the configuration settings for the SNVS_LP peripheral. To initialize this structure to reasonable defaults, call the SNVS_LP_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 srtcCalEnable

true: SRTC calibration mechanism is enabled; false: No calibration is used

uint32_t srtcCalValue

Defines signed calibration value for SRTC; This is a 5-bit 2's complement value, range from -16 to +15

2.56 SPDIF: Sony/Philips Digital Interface

void SPDIF_Init(SPDIF_Type *base, const *spdif_config_t* *config)

Initializes the SPDIF peripheral.

Ungates the SPDIF clock, resets the module, and configures SPDIF with a configuration structure. The configuration structure can be custom filled or set with default values by SPDIF_GetDefaultConfig().

Note: This API should be called at the beginning of the application to use the SPDIF driver. Otherwise, accessing the SPDIF module can cause a hard fault because the clock is not enabled.

Parameters

- base – SPDIF base pointer
- config – SPDIF configuration structure.

void SPDIF_GetDefaultConfig(*spdif_config_t* *config)

Sets the SPDIF configuration structure to default values.

This API initializes the configuration structure for use in SPDIF_Init. The initialized structure can remain unchanged in SPDIF_Init, or it can be modified before calling SPDIF_Init. This is an example.

```
spdif_config_t config;
SPDIF_GetDefaultConfig(&config);
```

Parameters

- config – pointer to master configuration structure

void SPDIF_Deinit(SPDIF_Type *base)

De-initializes the SPDIF peripheral.

This API gates the SPDIF clock. The SPDIF module can't operate unless SPDIF_Init is called to enable the clock.

Parameters

- base – SPDIF base pointer

uint32_t SPDIF_GetInstance(SPDIF_Type *base)

Get the instance number for SPDIF.

Parameters

- base – SPDIF base pointer.

static inline void SPDIF_TxFIFOReset(SPDIF_Type *base)

Resets the SPDIF Tx.

This function makes Tx FIFO in reset mode.

Parameters

- base – SPDIF base pointer

static inline void SPDIF_RxFIFOReset(SPDIF_Type *base)

Resets the SPDIF Rx.

This function enables the software reset and FIFO reset of SPDIF Rx. After reset, clear the reset bit.

Parameters

- base – SPDIF base pointer

void SPDIF_TxEnable(SPDIF_Type *base, bool enable)

Enables/disables the SPDIF Tx.

Parameters

- base – SPDIF base pointer
- enable – True means enable SPDIF Tx, false means disable.

static inline void SPDIF_RxEnable(SPDIF_Type *base, bool enable)

Enables/disables the SPDIF Rx.

Parameters

- base – SPDIF base pointer
- enable – True means enable SPDIF Rx, false means disable.

static inline uint32_t SPDIF_GetStatusFlag(SPDIF_Type *base)

Gets the SPDIF status flag state.

Parameters

- base – SPDIF base pointer

Returns

SPDIF status flag value. Use the `_spdif_interrupt_enable_t` to get the status value needed.

static inline void SPDIF_ClearStatusFlags(SPDIF_Type *base, uint32_t mask)

Clears the SPDIF status flag state.

Parameters

- base – SPDIF base pointer
- mask – State mask. It can be a combination of the `_spdif_interrupt_enable_t` member. Notice these members cannot be included, as these flags cannot be cleared by writing 1 to these bits:
 - kSPDIF_UChannelReceiveRegisterFull
 - kSPDIF_QChannelReceiveRegisterFull
 - kSPDIF_TxFIFOEmpty
 - kSPDIF_RxFIFOFull

static inline void SPDIF_EnableInterrupts(SPDIF_Type *base, uint32_t mask)

Enables the SPDIF Tx interrupt requests.

Parameters

- base – SPDIF base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSPDIF_WordStartInterruptEnable
 - kSPDIF_SyncErrorInterruptEnable
 - kSPDIF_FIFOWarningInterruptEnable
 - kSPDIF_FIFORequestInterruptEnable
 - kSPDIF_FIFOErrorInterruptEnable

static inline void SPDIF_DisableInterrupts(SPDIF_Type *base, uint32_t mask)

Disables the SPDIF Tx interrupt requests.

Parameters

- base – SPDIF base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSPDIF_WordStartInterruptEnable
 - kSPDIF_SyncErrorInterruptEnable
 - kSPDIF_FIFOWarningInterruptEnable
 - kSPDIF_FIFORequestInterruptEnable
 - kSPDIF_FIFOErrorInterruptEnable

```
static inline void SPDIF_EnableDMA(SPDIF_Type *base, uint32_t mask, bool enable)
```

Enables/disables the SPDIF DMA requests.

Parameters

- base – SPDIF base pointer
- mask – SPDIF DMA enable mask, The parameter can be a combination of the following sources if defined
 - kSPDIF_RxDMAEnable
 - kSPDIF_TxDMAEnable
- enable – True means enable DMA, false means disable DMA.

```
static inline uint32_t SPDIF_TxGetLeftDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Tx left data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF_TxGetRightDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Tx right data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF_RxGetLeftDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Rx left data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF_RxGetRightDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Rx right data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
void SPDIF_TxSetSampleRate(SPDIF_Type *base, uint32_t sampleRate_Hz, uint32_t  
                           sourceClockFreq_Hz)
```

Configures the SPDIF Tx sample rate.

The audio format can be changed at run-time. This function configures the sample rate.

Parameters

- base – SPDIF base pointer.
- sampleRate_Hz – SPDIF sample rate frequency in Hz.
- sourceClockFreq_Hz – SPDIF tx clock source frequency in Hz.

uint32_t SPDIF_GetRxSampleRate(SPDIF_Type *base, uint32_t clockSourceFreq_Hz)

Configures the SPDIF Rx audio format.

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – SPDIF base pointer.
- clockSourceFreq_Hz – SPDIF system clock frequency in Hz.

void SPDIF_WriteBlocking(SPDIF_Type *base, uint8_t *buffer, uint32_t size)

Sends data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SPDIF base pointer.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

static inline void SPDIF_WriteLeftData(SPDIF_Type *base, uint32_t data)

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

static inline void SPDIF_WriteRightData(SPDIF_Type *base, uint32_t data)

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

static inline void SPDIF_WriteChannelStatusHigh(SPDIF_Type *base, uint32_t data)

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

static inline void SPDIF_WriteChannelStatusLow(SPDIF_Type *base, uint32_t data)

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

`void SPDIF_ReadBlocking(SPDIF_Type *base, uint8_t *buffer, uint32_t size)`

Receives data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- `base` – SPDIF base pointer.
- `buffer` – Pointer to the data to be read.
- `size` – Bytes to be read.

`static inline uint32_t SPDIF_ReadLeftData(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- `base` – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

`static inline uint32_t SPDIF_ReadRightData(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- `base` – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

`static inline uint32_t SPDIF_ReadChannelStatusHigh(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- `base` – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

`static inline uint32_t SPDIF_ReadChannelStatusLow(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- `base` – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

`static inline uint32_t SPDIF_ReadQChannel(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- `base` – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

`static inline uint32_t SPDIF_ReadUChannel(SPDIF_Type *base)`

Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

```
void SPDIF__TransferTxCreateHandle(SPDIF_Type *base, spdif_handle_t *handle,  
                                  spdif_transfer_callback_t callback, void *userData)
```

Initializes the SPDIF Tx handle.

This function initializes the Tx handle for the SPDIF Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SPDIF base pointer
- handle – SPDIF handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function

```
void SPDIF__TransferRxCreateHandle(SPDIF_Type *base, spdif_handle_t *handle,  
                                  spdif_transfer_callback_t callback, void *userData)
```

Initializes the SPDIF Rx handle.

This function initializes the Rx handle for the SPDIF Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SPDIF base pointer.
- handle – SPDIF handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function.

```
status_t SPDIF__TransferSendNonBlocking(SPDIF_Type *base, spdif_handle_t *handle,  
                                         spdif_transfer_t *xfer)
```

Performs an interrupt non-blocking send transfer on SPDIF.

Note: This API returns immediately after the transfer initiates. Call the `SPDIF_TxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SPDIF_Busy`, the transfer is finished.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the `spdif_handle_t` structure which stores the transfer state.
- xfer – Pointer to the `spdif_transfer_t` structure.

Return values

- `kStatus__Success` – Successfully started the data receive.
- `kStatus_SPDIF__TxBusy` – Previous receive still not finished.
- `kStatus__InvalidArgument` – The input parameter is invalid.

status_t SPDIF__TransferReceiveNonBlocking(SPDIF_Type *base, *spdif_handle_t* *handle, *spdif_transfer_t* *xfer)

Performs an interrupt non-blocking receive transfer on SPDIF.

Note: This API returns immediately after the transfer initiates. Call the SPDIF_RxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SPDIF_Busy, the transfer is finished.

Parameters

- base – SPDIF base pointer
- handle – Pointer to the *spdif_handle_t* structure which stores the transfer state.
- xfer – Pointer to the *spdif_transfer_t* structure.

Return values

- kStatus__Success – Successfully started the data receive.
- kStatus_SPDIF_RxBusy – Previous receive still not finished.
- kStatus_InvalidArgument – The input parameter is invalid.

status_t SPDIF__TransferGetSendCount(SPDIF_Type *base, *spdif_handle_t* *handle, *size_t* *count)

Gets a set byte count.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the *spdif_handle_t* structure which stores the transfer state.
- count – Bytes count sent.

Return values

- kStatus__Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

status_t SPDIF__TransferGetReceiveCount(SPDIF_Type *base, *spdif_handle_t* *handle, *size_t* *count)

Gets a received byte count.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the *spdif_handle_t* structure which stores the transfer state.
- count – Bytes count received.

Return values

- kStatus__Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void SPDIF__TransferAbortSend(SPDIF_Type *base, *spdif_handle_t* *handle)

Aborts the current send.

Note: This API can be called any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the *spdif_handle_t* structure which stores the transfer state.

void SPDIF__TransferAbortReceive(SPDIF_Type *base, *spdif_handle_t* *handle)

Aborts the current IRQ receive.

Note: This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – SPDIF base pointer
- handle – Pointer to the *spdif_handle_t* structure which stores the transfer state.

void SPDIF__TransferTxHandleIRQ(SPDIF_Type *base, *spdif_handle_t* *handle)

Tx interrupt handler.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the *spdif_handle_t* structure.

void SPDIF__TransferRxHandleIRQ(SPDIF_Type *base, *spdif_handle_t* *handle)

Rx interrupt handler.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the *spdif_handle_t* structure.

FSL_SPDIF_DRIVER_VERSION

Version 2.0.7

SPDIF return status.

Values:

enumerator kStatus_SPDIF_RxDPLLLocked

SPDIF Rx PLL locked.

enumerator kStatus_SPDIF_TxFIFOError

SPDIF Tx FIFO error.

enumerator kStatus_SPDIF_TxFIFOResync

SPDIF Tx left and right FIFO resync.

enumerator kStatus_SPDIF_RxCnew
SPDIF Rx status channel value updated.

enumerator kStatus_SPDIF_ValidatyNoGood
SPDIF validaty flag not good.

enumerator kStatus_SPDIF_RxIllegalSymbol
SPDIF Rx receive illegal symbol.

enumerator kStatus_SPDIF_RxParityBitError
SPDIF Rx parity bit error.

enumerator kStatus_SPDIF_UChannelOverrun
SPDIF receive U channel overrun.

enumerator kStatus_SPDIF_QChannelOverrun
SPDIF receive Q channel overrun.

enumerator kStatus_SPDIF_UQChannelSync
SPDIF U/Q channel sync found.

enumerator kStatus_SPDIF_UQChannelFrameError
SPDIF U/Q channel frame error.

enumerator kStatus_SPDIF_RxFIFOError
SPDIF Rx FIFO error.

enumerator kStatus_SPDIF_RxFIFOResync
SPDIF Rx left and right FIFO resync.

enumerator kStatus_SPDIF_LockLoss
SPDIF Rx PLL clock lock loss.

enumerator kStatus_SPDIF_TxIdle
SPDIF Tx is idle

enumerator kStatus_SPDIF_RxIdle
SPDIF Rx is idle

enumerator kStatus_SPDIF_QueueFull
SPDIF queue full

enum _spdif_rxfull_select

SPDIF Rx FIFO full falg select, it decides when assert the rx full flag.

Values:

enumerator kSPDIF_RxFull1Sample
Rx full at least 1 sample in left and right FIFO

enumerator kSPDIF_RxFull4Samples
Rx full at least 4 sample in left and right FIFO

enumerator kSPDIF_RxFull8Samples
Rx full at least 8 sample in left and right FIFO

enumerator kSPDIF_RxFull16Samples
Rx full at least 16 sample in left and right FIFO

enum _spdif_txempty_select

SPDIF tx FIFO EMPTY falg select, it decides when assert the tx empty flag.

Values:

enumerator kSPDIF__TxEmpty0Sample

Tx empty at most 0 sample in left and right FIFO

enumerator kSPDIF__TxEmpty4Samples

Tx empty at most 4 sample in left and right FIFO

enumerator kSPDIF__TxEmpty8Samples

Tx empty at most 8 sample in left and right FIFO

enumerator kSPDIF__TxEmpty12Samples

Tx empty at most 12 sample in left and right FIFO

enum __spdif__uchannel__source

SPDIF U channel source.

Values:

enumerator kSPDIF__NoUChannel

No embedded U channel

enumerator kSPDIF__UChannelFromRx

U channel from receiver, it is CD mode

enumerator kSPDIF__UChannelFromTx

U channel from on chip tx

enum __spdif__gain__select

SPDIF clock gain.

Values:

enumerator kSPDIF__GAIN__24

Gain select is 24

enumerator kSPDIF__GAIN__16

Gain select is 16

enumerator kSPDIF__GAIN__12

Gain select is 12

enumerator kSPDIF__GAIN__8

Gain select is 8

enumerator kSPDIF__GAIN__6

Gain select is 6

enumerator kSPDIF__GAIN__4

Gain select is 4

enumerator kSPDIF__GAIN__3

Gain select is 3

enum __spdif__tx__source

SPDIF tx data source.

Values:

enumerator kSPDIF__txFromReceiver

Tx data directly through SPDIF receiver

enumerator kSPDIF__txNormal

Normal operation, data from processor

enum _spdif_validity_config

SPDIF tx data source.

Values:

enumerator kSPDIF__validityFlagAlwaysSet

Outgoing validity flags always set

enumerator kSPDIF__validityFlagAlwaysClear

Outgoing validity flags always clear

The SPDIF interrupt enable flag.

Values:

enumerator kSPDIF__RxDPLLLocked

SPDIF DPLL locked

enumerator kSPDIF__TxFIFOError

Tx FIFO underrun or overrun

enumerator kSPDIF__TxFIFOResync

Tx FIFO left and right channel resync

enumerator kSPDIF__RxControlChannelChange

SPDIF Rx control channel value changed

enumerator kSPDIF__ValidityFlagNoGood

SPDIF validity flag no good

enumerator kSPDIF__RxIllegalSymbol

SPDIF receiver found illegal symbol

enumerator kSPDIF__RxParityBitError

SPDIF receiver found parity bit error

enumerator kSPDIF__UChannelReceiveRegisterFull

SPDIF U channel receive register full

enumerator kSPDIF__UChannelReceiveRegisterOverrun

SPDIF U channel receive register overrun

enumerator kSPDIF__QChannelReceiveRegisterFull

SPDIF Q channel receive register full

enumerator kSPDIF__QChannelReceiveRegisterOverrun

SPDIF Q channel receive register overrun

enumerator kSPDIF__UQChannelSync

SPDIF U/Q channel sync found

enumerator kSPDIF__UQChannelFrameError

SPDIF U/Q channel frame error

enumerator kSPDIF__RxFIFOError

SPDIF Rx FIFO underrun/overrun

enumerator kSPDIF__RxFIFOResync

SPDIF Rx left and right FIFO resync

enumerator kSPDIF__LockLoss

SPDIF receiver loss of lock

enumerator kSPDIF_TxFIFOEmpty
SPDIF Tx FIFO empty

enumerator kSPDIF_RxFIFOFull
SPDIF Rx FIFO full

enumerator kSPDIF_AllInterrupt
all interrupt

The DMA request sources.

Values:

enumerator kSPDIF_RxDMAEnable
Rx FIFO full

enumerator kSPDIF_TxDMAEnable
Tx FIFO empty

typedef enum *_spdif_rxfull_select* spdif_rxfull_select_t
SPDIF Rx FIFO full falg select, it decides when assert the rx full flag.

typedef enum *_spdif_txempty_select* spdif_txempty_select_t
SPDIF tx FIFO EMPTY falg select, it decides when assert the tx empty flag.

typedef enum *_spdif_uchannel_source* spdif_uchannel_source_t
SPDIF U channel source.

typedef enum *_spdif_gain_select* spdif_gain_select_t
SPDIF clock gain.

typedef enum *_spdif_tx_source* spdif_tx_source_t
SPDIF tx data source.

typedef enum *_spdif_validity_config* spdif_validity_config_t
SPDIF tx data source.

typedef struct *_spdif_config* spdif_config_t
SPDIF user configuration structure.

typedef struct *_spdif_transfer* spdif_transfer_t
SPDIF transfer structure.

typedef struct *_spdif_handle* spdif_handle_t

typedef void (*spdif_transfer_callback_t)(SPDIF_Type *base, *spdif_handle_t* *handle, *status_t* status, void *userData)

SPDIF transfer callback prototype.

SPDIF_XFER_QUEUE_SIZE
SPDIF transfer queue size, user can refine it according to use case.

struct *_spdif_config*
#include <fsl_spdif.h> SPDIF user configuration structure.

Public Members

bool isTxAutoSync
If auto sync mechanism open

`bool isRxAutoSync`
 If auto sync mechanism open

`uint8_t DPLLClkSource`
 SPDIF DPLL clock source, range from 0~15, meaning is chip-specific

`uint8_t txClkSource`
 SPDIF tx clock source, range from 0~7, meaning is chip-specific

`spdif_rxfull_select_t rxFullSelect`
 SPDIF rx buffer full select

`spdif_txempty_select_t txFullSelect`
 SPDIF tx buffer empty select

`spdif_uchannel_source_t uChannelSrc`
 U channel source

`spdif_tx_source_t txSource`
 SPDIF tx data source

`spdif_validity_config_t validityConfig`
 Validity flag config

`spdif_gain_select_t gain`
 Rx receive clock measure gain parameter.

`struct _spdif_transfer`
`#include <fsl_spdif.h>` SPDIF transfer structure.

Public Members

`uint8_t *data`
 Data start address to transfer.

`uint8_t *qdata`
 Data buffer for Q channel

`uint8_t *udata`
 Data buffer for C channel

`size_t dataSize`
 Transfer size.

`struct _spdif_handle`
`#include <fsl_spdif.h>` SPDIF handle structure.

Public Members

`uint32_t state`
 Transfer status

`spdif_transfer_callback_t callback`
 Callback function called at transfer event

`void *userData`
 Callback parameter passed to callback function

`spdif_transfer_t spdifQueue[(4U)]`
 Transfer queue storing queued transfer

`size_t transferSize[(4U)]`
Data bytes need to transfer

`volatile uint8_t queueUser`
Index for user to queue transfer

`volatile uint8_t queueDriver`
Index for driver to get the transfer data and size

`uint8_t watermark`
Watermark value

2.57 SPDIF eDMA Driver

```
void SPDIF__TransferTxCreateHandleEDMA(SPDIF_Type *base, spdif_edma_handle_t *handle,  
                                       spdif_edma_callback_t callback, void *userData,  
                                       edma_handle_t *dmaLeftHandle, edma_handle_t  
                                       *dmaRightHandle)
```

Initializes the SPDIF eDMA handle.

This function initializes the SPDIF master DMA handle, which can be used for other SPDIF master transactional APIs. Usually, for a specified SPDIF instance, call this API once to get the initialized handle.

Parameters

- `base` – SPDIF base pointer.
- `handle` – SPDIF eDMA handle pointer.
- `callback` – Pointer to user callback function.
- `userData` – User parameter passed to the callback function.
- `dmaLeftHandle` – eDMA handle pointer for left channel, this handle shall be static allocated by users.
- `dmaRightHandle` – eDMA handle pointer for right channel, this handle shall be static allocated by users.

```
void SPDIF__TransferRxCreateHandleEDMA(SPDIF_Type *base, spdif_edma_handle_t *handle,  
                                       spdif_edma_callback_t callback, void *userData,  
                                       edma_handle_t *dmaLeftHandle, edma_handle_t  
                                       *dmaRightHandle)
```

Initializes the SPDIF Rx eDMA handle.

This function initializes the SPDIF slave DMA handle, which can be used for other SPDIF master transactional APIs. Usually, for a specified SPDIF instance, call this API once to get the initialized handle.

Parameters

- `base` – SPDIF base pointer.
- `handle` – SPDIF eDMA handle pointer.
- `callback` – Pointer to user callback function.
- `userData` – User parameter passed to the callback function.
- `dmaLeftHandle` – eDMA handle pointer for left channel, this handle shall be static allocated by users.
- `dmaRightHandle` – eDMA handle pointer for right channel, this handle shall be static allocated by users.

status_t SPDIF__TransferSendEDMA(SPDIF_Type *base, *spdif_edma_handle_t* *handle, *spdif_edma_transfer_t* *xfer)

Performs a non-blocking SPDIF transfer using DMA.

Note: This interface returns immediately after the transfer initiates. Call SPDIF_GetTransferStatus to poll the transfer status and check whether the SPDIF transfer is finished.

Parameters

- base – SPDIF base pointer.
- handle – SPDIF eDMA handle pointer.
- xfer – Pointer to the DMA transfer structure.

Return values

- kStatus__Success – Start a SPDIF eDMA send successfully.
- kStatus__InvalidArgument – The input argument is invalid.
- kStatus__TxBusy – SPDIF is busy sending data.

status_t SPDIF__TransferReceiveEDMA(SPDIF_Type *base, *spdif_edma_handle_t* *handle, *spdif_edma_transfer_t* *xfer)

Performs a non-blocking SPDIF receive using eDMA.

Note: This interface returns immediately after the transfer initiates. Call the SPDIF_GetReceiveRemainingBytes to poll the transfer status and check whether the SPDIF transfer is finished.

Parameters

- base – SPDIF base pointer
- handle – SPDIF eDMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- kStatus__Success – Start a SPDIF eDMA receive successfully.
- kStatus__InvalidArgument – The input argument is invalid.
- kStatus__RxBusy – SPDIF is busy receiving data.

void SPDIF__TransferAbortSendEDMA(SPDIF_Type *base, *spdif_edma_handle_t* *handle)

Aborts a SPDIF transfer using eDMA.

Parameters

- base – SPDIF base pointer.
- handle – SPDIF eDMA handle pointer.

void SPDIF__TransferAbortReceiveEDMA(SPDIF_Type *base, *spdif_edma_handle_t* *handle)

Aborts a SPDIF receive using eDMA.

Parameters

- base – SPDIF base pointer
- handle – SPDIF eDMA handle pointer.

```
status_t SPDIF_TransferGetSendCountEDMA(SPDIF_Type *base, spdif_edma_handle_t *handle,
                                         size_t *count)
```

Gets byte count sent by SPDIF.

Parameters

- base – SPDIF base pointer.
- handle – SPDIF eDMA handle pointer.
- count – Bytes count sent by SPDIF.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is no non-blocking transaction in progress.

```
status_t SPDIF_TransferGetReceiveCountEDMA(SPDIF_Type *base, spdif_edma_handle_t
                                           *handle, size_t *count)
```

Gets byte count received by SPDIF.

Parameters

- base – SPDIF base pointer
- handle – SPDIF eDMA handle pointer.
- count – Bytes count received by SPDIF.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is no non-blocking transaction in progress.

```
FSL_SPDIF_EDMA_DRIVER_VERSION
```

Version 2.0.8

```
typedef struct _spdif_edma_handle spdif_edma_handle_t
```

```
typedef void (*spdif_edma_callback_t)(SPDIF_Type *base, spdif_edma_handle_t *handle, status_t
status, void *userData)
```

SPDIF eDMA transfer callback function for finish and error.

```
typedef struct _spdif_edma_transfer spdif_edma_transfer_t
```

SPDIF transfer structure.

```
struct _spdif_edma_transfer
```

```
#include <fsl_spdif_edma.h> SPDIF transfer structure.
```

Public Members

```
uint8_t *leftData
```

Data start address to transfer.

```
uint8_t *rightData
```

Data start address to transfer.

```
size_t dataSize
```

Transfer size.

```
struct _spdif_edma_handle
```

```
#include <fsl_spdif_edma.h> SPDIF DMA transfer handle, users should not touch the content
of the handle.
```

Public Members

edma_handle_t *dmaLeftHandle
DMA handler for SPDIF left channel

edma_handle_t *dmaRightHandle
DMA handler for SPDIF right channel

uint8_t nbytes
eDMA minor byte transfer count initially configured.

uint8_t count
The transfer data count in a DMA request

uint32_t state
Internal state for SPDIF eDMA transfer

spdif_edma_callback_t callback
Callback for users while transfer finish or error occurs

void *userData
User callback parameter

edma_tcd_t leftTcd[(4U) + 1U]
TCD pool for eDMA transfer.

edma_tcd_t rightTcd[(4U) + 1U]
TCD pool for eDMA transfer.

spdif_edma_transfer_t spdifQueue[(4U)]
Transfer queue storing queued transfer.

size_t transferSize[(4U)]
Data bytes need to transfer, left and right are the same, so use one

volatile uint8_t queueUser
Index for user to queue transfer.

volatile uint8_t queueDriver
Index for driver to get the transfer data and size

2.58 SRC: System Reset Controller Driver

FSL_SRC_DRIVER_VERSION

SRC driver version 2.0.1.

enum _src_reset_status_flags

SRC reset status flags.

Values:

enumerator kSRC_ResetOutputEnableFlag

This bit indicates if RESET status is driven out on PTE0 pin.

enumerator kSRC_TemperatureSensorResetFlag

Indicates whether the reset was the result of software reset from on-chip Temperature Sensor. Temperature Sensor Interrupt needs to be served before this bit can be cleaned.

enumerator kSRC_Wdog3ResetFlag

IC Watchdog3 Time-out reset. Indicates whether the reset was the result of the watchdog3 time-out event.

enumerator kSRC_SoftwareResetFlag

Indicates a reset has been caused by software setting of SYSRESETREQ bit in Application Interrupt and Reset Control Register in the ARM core.

enumerator kSRC_JTAGSystemResetFlag

Indicates whether the reset was the result of software reset from JTAG

enumerator kSRC_JTAGSoftwareResetFlag

Indicates whether the reset was the result of setting SJC_GPCCR bit 31.

enumerator kSRC_JTAGGeneratedResetFlag

Indicates a reset has been caused by JTAG selection of certain IR codes: EXTEST or HIGHZ.

enumerator kSRC_WatchdogResetFlag

Indicates a reset has been caused by the watchdog timer timing out. This reset source can be blocked by disabling the watchdog.

enumerator kSRC_IppUserResetFlag

Indicates whether the reset was the result of the ipp_user_reset_b qualified reset.

enumerator kSRC_SNVSFailResetFlag

SNVS hardware failure will always cause a cold reset. This flag indicates whether the reset is a result of SNVS hardware failure.

enumerator kSRC_CsuResetFlag

Indicates whether the reset was the result of the csu_reset_b input.

enumerator kSRC_CoreLockupResetFlag

Indicates a reset has been caused by the ARM core indication of a LOCKUP event.

enumerator kSRC_PowerOnResetFlag

Indicates a reset has been caused by the power-on detection logic.

enumerator kSRC_LockupSysResetFlag

Indicates a reset has been caused by CPU lockup or software setting of SYSRESETREQ bit in Application Interrupt and Reset Control Register of the ARM core.

enumerator kSRC_IppResetPinFlag

Indicates whether reset was the result of ipp_reset_b pin (Power-up sequence).

enum _src_status_flags

SRC interrupt status flag.

Values:

enumerator kSRC_Core0WdogResetReqFlag

WDOG reset request from core0. Read-only status bit.

enum _src_mix_reset_stretch_cycles

Selection of SoC mix power reset stretch.

This type defines the SoC mix (Audio, ENET, uSDHC, EIM, QSPI, OCRAM, MMDC, etc) power up reset stretch mix reset width with the optional count of cycles

Values:

enumerator kSRC_MixResetStretchCycleAlt0

mix reset width is 1 x 88 ipg_cycle cycles.

enumerator kSRC_MixResetStretchCycleAlt1
mix reset width is 2 x 88 ipg_cycle cycles.

enumerator kSRC_MixResetStretchCycleAlt2
mix reset width is 3 x 88 ipg_cycle cycles.

enumerator kSRC_MixResetStretchCycleAlt3
mix reset width is 4 x 88 ipg_cycle cycles.

enum _src_wdog3_reset_option
Selection of WDOG3 reset option.

Values:

enumerator kSRC_Wdog3ResetOptionAlt0
Wdog3_rst_b asserts M4 reset (default).

enumerator kSRC_Wdog3ResetOptionAlt1
Wdog3_rst_b asserts global reset.

enum _src_warm_reset_bypass_count
Selection of WARM reset bypass count.

This type defines the 32KHz clock cycles to count before bypassing the MMDC acknowledge for WARM reset. If the MMDC acknowledge is not asserted before this counter is elapsed, a COLD reset will be initiated.

Values:

enumerator kSRC_WarmResetWaitAlways
System will wait until MMDC acknowledge is asserted.

enumerator kSRC_WarmResetWaitClk16
Wait 16 32KHz clock cycles before switching the reset.

enumerator kSRC_WarmResetWaitClk32
Wait 32 32KHz clock cycles before switching the reset.

enumerator kSRC_WarmResetWaitClk64
Wait 64 32KHz clock cycles before switching the reset.

typedef enum _src_mix_reset_stretch_cycles src_mix_reset_stretch_cycles_t
Selection of SoC mix power reset stretch.

This type defines the SoC mix (Audio, ENET, uSDHC, EIM, QSPI, OCRAM, MMDC, etc) power up reset stretch mix reset width with the optional count of cycles

typedef enum _src_wdog3_reset_option src_wdog3_reset_option_t
Selection of WDOG3 reset option.

typedef enum _src_warm_reset_bypass_count src_warm_reset_bypass_count_t
Selection of WARM reset bypass count.

This type defines the 32KHz clock cycles to count before bypassing the MMDC acknowledge for WARM reset. If the MMDC acknowledge is not asserted before this counter is elapsed, a COLD reset will be initiated.

static inline void SRC_EnableWDOG3Reset(SRC_Type *base, bool enable)

Enable the WDOG3 reset.

The WDOG3 reset is enabled by default.

Parameters

- base – SRC peripheral base address.

- enable – Enable the reset or not.

```
static inline void SRC_SetMixResetStretchCycles(SRC_Type *base, src_mix_reset_stretch_cycles_t option)
```

Set the mix power up reset stretch mix reset width.

Parameters

- base – SRC peripheral base address.
- option – Setting option, see to src_mix_reset_stretch_cycles_t.

```
static inline void SRC_EnableCoreDebugResetAfterPowerGate(SRC_Type *base, bool enable)
```

Debug reset would be asserted after power gating event.

Parameters

- base – SRC peripheral base address.
- enable – Enable the reset or not.

```
static inline void SRC_SetWdog3ResetOption(SRC_Type *base, src_wdog3_reset_option_t option)
```

Set the Wdog3_rst_b option.

Parameters

- base – SRC peripheral base address.
- option – Setting option, see to src_wdog3_reset_option_t.

```
static inline void SRC_DoSoftwareResetARMCoreDebug(SRC_Type *base)
```

Software reset for debug of arm platform only.

Parameters

- base – SRC peripheral base address.

```
static inline bool SRC_GetSoftwareResetARMCoreDebugDone(SRC_Type *base)
```

Check if the software reset for debug of arm platform only is done.

Parameters

- base – SRC peripheral base address.

```
static inline void SRC_EnableTemperatureSensorReset(SRC_Type *base, bool enable)
```

Enable the temperature sensor reset.

The temperature sensor reset is enabled by default. When the sensor reset happens, an flag bit would be asserted. This flag bit can be cleared only by the hardware reset.

Parameters

- base – SRC peripheral base address.
- enable – Enable the reset or not.

```
static inline void SRC_DoSoftwareResetARMCore0(SRC_Type *base)
```

Do software reset the ARM core0 only.

Parameters

- base – SRC peripheral base address.

```
static inline bool SRC_GetSoftwareResetARMCore0Done(SRC_Type *base)
```

Check if the software for ARM core0 is done.

Parameters

- base – SRC peripheral base address.

Returns

If the reset is done.

```
static inline void SRC_DoSoftwareResetARMCORE(SRC_Type *base)
```

Do software reset for ARM core.

This function can be used to assert the ARM core reset. Once it is called, the reset process will begin. After the reset process is finished, the command bit would be self cleared.

Parameters

- base – SRC peripheral base address.

```
static inline bool SRC_GetSoftwareResetARMCOREDone(SRC_Type *base)
```

Check if the software for ARM core is done.

Parameters

- base – SRC peripheral base address.

Returns

If the reset is done.

```
static inline void SRC_AssertEIMReset(SRC_Type *base, bool enable)
```

Assert the EIM reset.

EIM reset is needed in order to reconfigure the EIM chip select. The software reset bit must de-asserted since this is not self-refresh.

Parameters

- base – SRC peripheral base address.
- enable – Make the assertion or not.

```
static inline void SRC_EnableWDOGReset(SRC_Type *base, bool enable)
```

Enable the WDOG Reset in SRC.

WDOG Reset is enabled in SRC by default. If the WDOG event to SRC is masked, it would not create a reset to the chip. During the time the WDOG event is masked, when the WDOG event flag is asserted, it would remain asserted regardless of servicing the WDOG module. The only way to clear that bit is the hardware reset.

Parameters

- base – SRC peripheral base address.
- enable – Enable the reset or not.

```
static inline void SRC_EnableLockupReset(SRC_Type *base, bool enable)
```

Enable the lockup reset.

Parameters

- base – SRC peripheral base address.
- enable – Enable the reset or not.

```
static inline void SRC_EnableCoreLockupReset(SRC_Type *base, bool enable)
```

Enable the core lockup reset.

When enable the core lockup reset, the system would be reset when core lockup event happens.

Parameters

- base – SRC peripheral base address.
- enable – Enable the reset or not.

static inline uint32_t SRC_GetStatusFlags(SRC_Type *base)

Get interrupt status flags.

Parameters

- base – SRC peripheral base address.

Returns

Mask value of status flags. See to `$_src_status_flags`.

static inline uint32_t SRC_GetBootModeWord1(SRC_Type *base)

Get the boot mode register 1 value.

The Boot Mode register contains bits that reflect the status of BOOT_CFGx pins of the chip. See to chip-specific document for detail information about value.

Parameters

- base – SRC peripheral base address.

Returns

status of BOOT_CFGx pins of the chip.

static inline uint32_t SRC_GetBootModeWord2(SRC_Type *base)

Get the boot mode register 2 value.

The Boot Mode register contains bits that reflect the status of BOOT_MODEx Pins and fuse values that controls boot of the chip. See to chip-specific document for detail information about value.

Parameters

- base – SRC peripheral base address.

Returns

status of BOOT_MODEx Pins and fuse values that controls boot of the chip.

static inline uint32_t SRC_GetResetStatusFlags(SRC_Type *base)

Get the status flags of SRC.

Parameters

- base – SRC peripheral base address.

Returns

Mask value of status flags, see to `_src_reset_status_flags`.

void SRC_ClearResetStatusFlags(SRC_Type *base, uint32_t flags)

Clear the status flags of SRC.

Parameters

- base – SRC peripheral base address.
- flags – value of status flags to be cleared, see to `_src_reset_status_flags`.

static inline void SRC_SetGeneralPurposeRegister(SRC_Type *base, uint32_t index, uint32_t value)

Set value to general purpose registers.

General purpose registers (GPRx) would hold the value during reset process. Wakeup function could be kept in these register. For example, the GPR1 holds the entry function for waking-up from Partial SLEEP mode while the GPR2 holds the argument. Other GPRx register would store the arbitray values.

Parameters

- base – SRC peripheral base address.

- `index` – The index of GPRx register array. Note index 0 reponses the GPR1 register.
- `value` – Setting value for GPRx register.

static inline uint32_t SRC_GetGeneralPurposeRegister(SRC_Type *base, uint32_t index)

Get the value from general purpose registers.

Parameters

- `base` – SRC peripheral base address.
- `index` – The index of GPRx register array. Note index 0 reponses the GPR1 register.

Returns

The setting value for GPRx register.

2.59 TEMPMON: Temperature Monitor Module

FSL_TEMPMON_DRIVER_VERSION

TEMPMON driver version.

enum __tempmon_alarm_mode

TEMPMON alarm mode.

Values:

enumerator kTEMPMON_HighAlarmMode

The high alarm temperature interrupt mode.

enumerator kTEMPMON_PanicAlarmMode

The panic alarm temperature interrupt mode.

enumerator kTEMPMON_LowAlarmMode

The low alarm temperature interrupt mode.

typedef struct *tempmon_config* tempmon_config_t

TEMPMON temperature structure.

typedef enum *tempmon_alarm_mode* tempmon_alarm_mode

TEMPMON alarm mode.

void TEMPMON_Init(TEMPMON_Type *base, const *tempmon_config_t* *config)

Initializes the TEMPMON module.

Parameters

- `base` – TEMPMON base pointer
- `config` – Pointer to configuration structure.

void TEMPMON_Deinit(TEMPMON_Type *base)

Deinitializes the TEMPMON module.

Parameters

- `base` – TEMPMON base pointer

void TEMPMON_GetDefaultConfig(*tempmon_config_t* *config)

Gets the default configuration structure.

This function initializes the TEMPMON configuration structure to a default value. The default values are: `tempmonConfig->frequency = 0x02U`; `tempmonConfig->highAlarmTemp = 44U`; `tempmonConfig->panicAlarmTemp = 90U`; `tempmonConfig->lowAlarmTemp = 39U`;

Parameters

- config – Pointer to a configuration structure.

static inline void TEMPMON_StartMeasure(TEMPMON_Type *base)
start the temperature measurement process.

Parameters

- base – TEMPMON base pointer.

static inline void TEMPMON_StopMeasure(TEMPMON_Type *base)
stop the measurement process.

Parameters

- base – TEMPMON base pointer

float TEMPMON_GetCurrentTemperature(TEMPMON_Type *base)
Get current temperature with the fused temperature calibration data.

Parameters

- base – TEMPMON base pointer

Returns

current temperature with degrees Celsius.

void TEMPMON_SetTempAlarm(TEMPMON_Type *base, int16_t tempVal,
tempmon_alarm_mode alarmMode)

Set the temperature count (raw sensor output) that will generate an alarm interrupt.

Parameters

- base – TEMPMON base pointer
- tempVal – The alarm temperature with degrees Celsius
- alarmMode – The alarm mode.

TEMPMON_HOTTEMPMASK
TEMPMON calibration data mask.

TEMPMON_HOTTEMPSHIFT

TEMPMON_HOTCOUNTMASK

TEMPMON_HOTCOUNTSHIFT

TEMPMON_ROOMCOUNTMASK

TEMPMON_ROOMCOUNTSHIFT

struct __tempmon_config
#include <fsl_tempmon.h> TEMPMON temperature structure.

Public Members

uint16_t frequency
The temperature measure frequency.

int16_t highAlarmTemp
The high alarm temperature.

int16_t panicAlarmTemp
The panic alarm temperature.

int16_t lowAlarmTemp

The low alarm temperature.

2.60 TRNG: True Random Number Generator

FSL_TRNG_DRIVER_VERSION

TRNG driver version 2.0.19.

Current version: 2.0.19

Change log:

- version 2.0.19
 - Added support for MCXA and MCXL.
- 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/zeros 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.

2.61 WDOG: Watchdog Timer Driver

`void WDOG_GetDefaultConfig(wdog_config_t *config)`

Initializes the WDOG configuration structure.

This function initializes the WDOG configuration structure to default values. The default values are as follows.

```
wdogConfig->enableWdog = true;
wdogConfig->workMode.enableWait = true;
wdogConfig->workMode.enableStop = true;
wdogConfig->workMode.enableDebug = true;
wdogConfig->enableInterrupt = false;
wdogConfig->enablePowerdown = false;
wdogConfig->resetExtension = false;
wdogConfig->timeoutValue = 0xFFU;
wdogConfig->interruptTimeValue = 0x04u;
```

See also:

`wdog_config_t`

Parameters

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

`void WDOG_Init(WDOG_Type *base, const wdog_config_t *config)`

Initializes the WDOG.

This function initializes the WDOG. When called, the WDOG runs according to the configuration.

This is an example.

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0xffU;
config->interruptTimeValue = 0x04u;
WDOG_Init(wdog_base,&config);
```

Parameters

- `base` – WDOG peripheral base address
- `config` – The configuration of WDOG

`void WDOG_Deinit(WDOG_Type *base)`

Shuts down the WDOG.

This function shuts down the WDOG. Watchdog Enable bit is a write one once only bit. It is not possible to clear this bit by a software write, once the bit is set. This bit(WDE) can be set/reset only in debug mode(exception).

`static inline void WDOG_Enable(WDOG_Type *base)`

Enables the WDOG module.

This function writes a value into the WDOG_WCR register to enable the WDOG. This is a write one once only bit. It is not possible to clear this bit by a software write, once the bit is set. only debug mode exception.

Parameters

- `base` – WDOG peripheral base address

`static inline void WDOG_Disable(WDOG_Type *base)`

Disables the WDOG module.

This function writes a value into the WDOG_WCR register to disable the WDOG. This is a write one once only bit. It is not possible to clear this bit by a software write,once the bit is set. only debug mode exception

Parameters

- `base` – WDOG peripheral base address

`static inline void WDOG_TriggerSystemSoftwareReset(WDOG_Type *base)`

Trigger the system software reset.

This function will write to the WCR[SRS] bit to trigger a software system reset. This bit will automatically resets to “1” after it has been asserted to “0”. Note: Calling this API will reset the system right now, please using it with more attention.

Parameters

- `base` – WDOG peripheral base address

`static inline void WDOG_TriggerSoftwareSignal(WDOG_Type *base)`

Trigger an output assertion.

This function will write to the WCR[WDA] bit to trigger WDOG_B signal assertion. The WDOG_B signal can be routed to external pin of the chip, the output pin will turn to assertion along with WDOG_B signal. Note: The WDOG_B signal will remain assert until a power on reset occurred, so, please take more attention while calling it.

Parameters

- base – WDOG peripheral base address

```
static inline void WDOG_EnableInterrupts(WDOG_Type *base, uint16_t mask)
```

Enables the WDOG interrupt.

This bit is a write once only bit. Once the software does a write access to this bit, it will get locked and cannot be reprogrammed until the next system reset assertion

Parameters

- base – WDOG peripheral base address
- mask – The interrupts to enable The parameter can be combination of the following source if defined.
 - kWDOG_InterruptEnable

```
uint16_t WDOG_GetStatusFlags(WDOG_Type *base)
```

Gets the WDOG all reset status flags.

This function gets all reset status flags.

```
uint16_t status;
status = WDOG_GetStatusFlags (wdog_base);
```

See also:

`_wdog_status_flags`

- true: a related status flag has been set.
- false: a related status flag is not set.

Parameters

- base – WDOG peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

```
void WDOG_ClearInterruptStatus(WDOG_Type *base, uint16_t mask)
```

Clears the WDOG flag.

This function clears the WDOG status flag.

This is an example for clearing the interrupt flag.

```
WDOG_ClearStatusFlags(wdog_base, KWDOG_InterruptFlag);
```

Parameters

- base – WDOG peripheral base address
- mask – The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag

static inline void WDOG_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)

Sets the WDOG timeout value.

This function sets the timeout value. This function writes a value into WCR registers. The time-out value can be written at any point of time but it is loaded to the counter at the time when WDOG is enabled or after the service routine has been performed.

Parameters

- base – WDOG peripheral base address
- timeoutCount – WDOG timeout value; count of WDOG clock tick.

static inline void WDOG_SetInterruptTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)

Sets the WDOG interrupt count timeout value.

This function sets the interrupt count timeout value. This function writes a value into WIC registers which are write-once. This field is write once only. Once the software does a write access to this field, it will get locked and cannot be reprogrammed until the next system reset assertion.

Parameters

- base – WDOG peripheral base address
- timeoutCount – WDOG timeout value; count of WDOG clock tick.

static inline void WDOG_DisablePowerDownEnable(WDOG_Type *base)

Disable the WDOG power down enable bit.

This function disable the WDOG power down enable(PDE). This function writes a value into WMCR registers which are write-once. This field is write once only. Once software sets this bit it cannot be reset until the next system reset.

Parameters

- base – WDOG peripheral base address

void WDOG_Refresh(WDOG_Type *base)

Refreshes the WDOG timer.

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG peripheral base address

FSL_WDOG_DRIVER_VERSION

Defines WDOG driver version.

WDOG_REFRESH_KEY

enum _wdog_interrupt_enable

WDOG interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the WDOG interrupt configurations.

Values:

enumerator kWDOG_InterruptEnable

WDOG timeout generates an interrupt before reset

enum _wdog_status_flags

WDOG status flags.

This structure contains the WDOG status flags for use in the WDOG functions.

Values:

enumerator `kWDOG_RunningFlag`
Running flag, set when WDOG is enabled

enumerator `kWDOG_PowerOnResetFlag`
Power On flag, set when reset is the result of a powerOnReset

enumerator `kWDOG_TimeoutResetFlag`
Timeout flag, set when reset is the result of a timeout

enumerator `kWDOG_SoftwareResetFlag`
Software flag, set when reset is the result of a software

enumerator `kWDOG_InterruptFlag`
interrupt flag, whether interrupt has occurred or not

typedef struct *wdog_work_mode* `wdog_work_mode_t`
Defines WDOG work mode.

typedef struct *wdog_config* `wdog_config_t`
Describes WDOG configuration structure.

struct `_wdog_work_mode`
#include <fsl_wdog.h> Defines WDOG work mode.

Public Members

bool `enableWait`
If set to true, WDOG continues in wait mode

bool `enableStop`
If set to true, WDOG continues in stop mode

bool `enableDebug`
If set to true, WDOG continues in debug mode

struct `_wdog_config`
#include <fsl_wdog.h> Describes WDOG configuration structure.

Public Members

bool `enableWdog`
Enables or disables WDOG

wdog_work_mode_t `workMode`
Configures WDOG work mode in debug stop and wait mode

bool `enableInterrupt`
Enables or disables WDOG interrupt

uint16_t `timeoutValue`
Timeout value

uint16_t `interruptTimeValue`
Interrupt count timeout value

bool `softwareResetExtension`
software reset extension

bool `enablePowerDown`
power down enable bit

bool enableTimeOutAssert

Enable WDOG_B timeout assertion.

2.62 XBARA: Inter-Peripheral Crossbar Switch

void XBARA_Init(XBARA_Type *base)

Initializes the XBARA module.

This function un-gates the XBARA clock.

Parameters

- base – XBARA peripheral address.

void XBARA_Deinit(XBARA_Type *base)

Shuts down the XBARA module.

This function disables XBARA clock.

Parameters

- base – XBARA peripheral address.

void XBARA_SetSignalsConnection(XBARA_Type *base, xbar_input_signal_t input,
xbar_output_signal_t output)

Sets a connection between the selected XBARA_IN[*] input and the XBARA_OUT[*] output signal.

This function connects the XBARA input to the selected XBARA output. If more than one XBARA module is available, only the inputs and outputs from the same module can be connected.

Example:

```
XBARA_SetSignalsConnection(XBARA, kXBARA_InputPIT_TRG0, kXBARA_  
↪OutputDMAMUX18);
```

Parameters

- base – XBARA peripheral address.
- input – XBARA input signal.
- output – XBARA output signal.

uint32_t XBARA_GetStatusFlags(XBARA_Type *base)

Gets the active edge detection status.

This function gets the active edge detect status of all XBARA_OUTs. If the active edge occurs, the return value is asserted. When the interrupt or the DMA functionality is enabled for the XBARA_OUTx, this field is 1 when the interrupt or DMA request is asserted and 0 when the interrupt or DMA request has been cleared.

Parameters

- base – XBARA peripheral address.

Returns

the mask of these status flag bits.

void XBARA_ClearStatusFlags(XBARA_Type *base, uint32_t mask)

Clears the edge detection status flags of relative mask.

Parameters

- base – XBARA peripheral address.
- mask – the status flags to clear.

```
void XBARA_SetOutputSignalConfig(XBARA_Type *base, xbar_output_signal_t output, const
                                xbara_control_config_t *controlConfig)
```

Configures the XBARA control register.

This function configures an XBARA control register. The active edge detection and the DMA/IRQ function on the corresponding XBARA output can be set.

Example:

```
xbara_control_config_t userConfig;
userConfig.activeEdge = kXBARA_EdgeRising;
userConfig.requestType = kXBARA_RequestInterruptEnable;
XBARA_SetOutputSignalConfig(XBARA, kXBARA_OutputDMAMUX18, &userConfig);
```

Parameters

- base – XBARA peripheral address.
- output – XBARA output number.
- controlConfig – Pointer to structure that keeps configuration of control register.

```
enum _xbara_active_edge
```

XBARA active edge for detection.

Values:

```
enumerator kXBARA_EdgeNone
```

Edge detection status bit never asserts.

```
enumerator kXBARA_EdgeRising
```

Edge detection status bit asserts on rising edges.

```
enumerator kXBARA_EdgeFalling
```

Edge detection status bit asserts on falling edges.

```
enumerator kXBARA_EdgeRisingAndFalling
```

Edge detection status bit asserts on rising and falling edges.

```
enum _xbar_request
```

Defines the XBARA DMA and interrupt configurations.

Values:

```
enumerator kXBARA_RequestDisable
```

Interrupt and DMA are disabled.

```
enumerator kXBARA_RequestDMAEnable
```

DMA enabled, interrupt disabled.

```
enumerator kXBARA_RequestInterruptEnable
```

Interrupt enabled, DMA disabled.

```
enum _xbara_status_flag_t
```

XBARA status flags.

This provides constants for the XBARA status flags for use in the XBARA functions.

Values:

enumerator `kXBARA_EdgeDetectionOut0`

`XBAR_OUT0` active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut1`

`XBAR_OUT1` active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut2`

`XBAR_OUT2` active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut3`

`XBAR_OUT3` active edge interrupt flag, sets when active edge detected.

typedef enum `_xbara_active_edge` `xbara_active_edge_t`

`XBARA` active edge for detection.

typedef enum `_xbar_request` `xbara_request_t`

Defines the `XBARA` DMA and interrupt configurations.

typedef enum `_xbara_status_flag_t` `xbara_status_flag_t`

`XBARA` status flags.

This provides constants for the `XBARA` status flags for use in the `XBARA` functions.

typedef struct `XBARAControlConfig` `xbara_control_config_t`

Defines the configuration structure of the `XBARA` control register.

This structure keeps the configuration of `XBARA` control register for one output. Control registers are available only for a few outputs. Not every `XBARA` module has control registers.

`FSL_XBARA_DRIVER_VERSION`

`XBARA_SELx(base, output)`

`XBARA_WR_SELx_SELx(base, input, output)`

`kXBARA_RequestInterruptEnalbe`

struct `XBARAControlConfig`

`#include <fsl_xbara.h>` Defines the configuration structure of the `XBARA` control register.

This structure keeps the configuration of `XBARA` control register for one output. Control registers are available only for a few outputs. Not every `XBARA` module has control registers.

Public Members

`xbara_active_edge_t` `activeEdge`

Active edge to be detected.

`xbara_request_t` `requestType`

Selects DMA/Interrupt request.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called “packet-driven BDM” interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT](<https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/>) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to [FreeMASTER community](#) or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a “middleware” component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pd_bdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER “middleware” driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample applications Zephyr sample applications demonstrate Kconfig and Device Tree configuration which configure the FreeMASTER middleware module for a selected communication option (Serial, CAN, Network or RTT).

Refer to *readme.md* files in each sample directory for description of configuration options required to implement FreeMASTER connectivity.

Description

This section shows how to add the FreeMASTER Communication Driver into application and how to configure the connection to the FreeMASTER visualization tool.

Features The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space.
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables.
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control an embedded application.

Board Detection The FreeMASTER protocol V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

Memory Read This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must be shorter than the MTU to fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The driver uses size-aligned operations to read the target memory (for example; uses proper read-word instruction when an address is aligned to 4 bytes).

Memory Write Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; uses proper write-word instruction when an address is aligned to 4 bytes).

Masked Memory Write To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

Oscilloscope The protocol and driver enables any number of variables to be read at once with a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

The driver can be configured to support any number of Oscilloscope instances and enable simultaneously running graphs to be displayed on the host computer screen.

Recorder The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

The driver can be configured to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables setting the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

TSA With the TSA feature, data types and variables can be described directly in the application source code. Such information is later provided to the FreeMASTER tool which may use it instead of reading symbol data from the application ELF executable file.

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

TSA Safety When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

Application commands The Application Commands are high-level messages that can be delivered from the PC Host to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing in general.

Pipes The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losses.

Serial single-wire operation The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- “External” single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- “True” single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as “automatic” in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the .c files must be added to the project, compiled, and linked together with the application.
 - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
 - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
 - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
 - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
 - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
 - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.
- **src/drivers/[sdk]/serial** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- **src/drivers/[sdk]/can** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- **src/drivers/[sdk]/network** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
 - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
 - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR    [0|1]
#define FMSTR_SHORT_INTR   [0|1]
#define FMSTR_POLL_DRIVEN [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See [Driver interrupt modes](#).

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- FMSTR_SERIAL - serial communication protocol
- FMSTR_CAN - using CAN communication
- FMSTR_PDBDM - using packet-driven BDM communication
- FMSTR_NET - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as **FMSTR_SERIAL_DRV**. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32...255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0...255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the “True” single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE

```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetCanBaseAddress()` to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options**FMSTR_DISABLE**

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+@W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options**FMSTR_USE_SCOPE**

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options**FMSTR_USE_RECORDER**

```
#define FMSTR_USE_RECORDER [number]
```


Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.
Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature. Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using FMSTR_RegisterAppCmdCall(). Default value is 0.

TSA options**FMSTR_USE_TSA**

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool. Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables. Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project. Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions. Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used. Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support. The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_QUEUE_SIZE*) and the character time is the time needed to transmit or receive a single byte over the SCI line.

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

In this mode, both the communication and the FreeMASTER protocol decoding are done in the *FMSTR_Poll* routine. No interrupts are needed and the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, and similar handlers compile to an empty code.

When using this mode, ensure that the *FMSTR_Poll* function is called by the application at least once per the serial "character time" which is the time needed to transmit or receive a single character.

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the *FMSTR_Poll* routine. An application interrupt can occur in the middle of the

Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

The same issue may appear even in the full interrupt mode (FMSTR_LONG_INTR), if volatile variables are modified in the interrupt code with a priority higher than the priority of the communication interrupt.

Data types Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the platform-specific header file.

To prevent name conflicts with the symbols used in the application, all data types, macros, and functions have the FMSTR_ prefix. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

Communication interface initialization The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is the application startup code responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

Note: It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

FreeMASTER Recorder calls When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere else. The example applications provided together with the driver code call the FMSTR_Recorder in the main application loop.

In applications where FMSTR_Recorder is called periodically with a constant period, specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

Driver usage Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all *.c files of the FreeMASTER driver from the `src/common/platforms/[your_platform]` folder are a part of the project. See [Driver files](#) for more details.
- Configure the FreeMASTER driver by creating or editing the `freemaster_cfg.h` file and by saving it into the application project directory. See [Driver configuration](#) for more details.
- Include the `freemaster.h` file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, call the FMSTR_Poll API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally so that the interrupts can be handled by the CPU.

Communication troubleshooting The most common problem that causes communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When a communication between the PC Host running FreeMASTER and the target MCU cannot be established, try enabling the FMSTR_DEBUG_TX option in the `freemaster_cfg.h` file and call the FMSTR_Poll function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Driver API

This section describes the driver Application Programmers' Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

Control API There are three key functions to initialize and use the driver.

FMSTR_Init

Prototype

```
FMSTR_BOOL FMSTR_Init(void);
```

- Declaration: `freemaster.h`
- Implementation: `freemaster_protocol.c`

Description This function initializes the internal variables of the FreeMASTER driver and enables the communication interface. This function does not change the configuration of the selected communication module. The hardware module must be initialized before the [FMSTR_Init](#) function is called.

A call to this function must occur before calling any other FreeMASTER driver API functions.

FMSTR_Poll

Prototype

```
void FMSTR_Poll(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see [Driver interrupt modes](#)). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the [FMSTR_Poll](#) function is called during the “idle” time in the main application task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the FMSTR_Poll function is called at least once per the time calculated as:

$$N * Tchar$$

where:

- *N* is equal to the length of the receive FIFO queue (configured by the FMSTR_COMM_RQUEUE_SIZE macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

Note: In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void);
void FMSTR_CanIsr(void);
```

- Declaration: *freemaster.h*
- Implementation: *hw-specific low-level driver C file*

Description This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see [Driver interrupt modes](#)), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

Note: In a poll-driven mode, this function is compiled as an empty function and does not have to be used.

Recorder API

FMSTR_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance 0 which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the *FMSTR_REC_BUFF_SIZE* and *FMSTR_REC_TIMEBASE* options.

For more information, see [Configurable items](#).

FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance *recIndex*. If the selected Recorder is not active when the *FMSTR_Recorder* function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down the follow-up samples (number of *FMSTR_Recorder* function calls) and de-activates the Recorder when the required post-trigger samples are finished.

The *FMSTR_Recorder* function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

FMSTR_RecorderTrigger

Prototype

```
void FMSTR_RecorderTrigger(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application code for programmatic control over the Recorder triggering. This can be useful when a more complex triggering conditions need to be used.

Fast Recorder API The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

TSA Tables When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), it defines the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always exactly one TSA Table List defined, which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. The symbols can then be used to create FreeMASTER variables for visualization or control.

TSA table definition The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide an access to the individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-language symbol.

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

After this opening macro, the TSA descriptors are placed using these macros:

```
/* Adding variable descriptors */
FMSTR_TSA_RW_VAR(name, type) /* read/write variable entry */
FMSTR_TSA_RO_VAR(name, type) /* read-only variable entry */

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
FMSTR_TSA_MEMBER(struct_name, member_name, type) /* structure member entry */

/* Memory blocks */
FMSTR_TSA_RW_MEM(name, type, address, size) /* read/write memory block */
FMSTR_TSA_RO_MEM(name, type, address, size) /* read-only memory block */
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

TSA descriptor parameters The TSA descriptor macros accept these parameters:

- *name* — variable name. The variable must be defined before the TSA descriptor references it.
- *type* — variable or member type. Only one of the pre-defined type constants may be used (see below).
- *struct_name* — structure type name. The type must be defined (typedef) before the TSA descriptor references it.

- *member_name* — structure member name.

Note: The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table.

Note: To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

TSA variable types The table lists *type* identifiers which can be used in TSA descriptors:

Constant	Description
FMSTR_TSA_UINTn	Unsigned integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size <i>n</i> bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(<i>m,n</i>)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(<i>m,n</i>)	Unsigned fractional number in general UQ form (m+n total bits)
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type
FMSTR_TSA_POINTER	Generic pointer type defined (platform-specific 16 or 32 bit)
FM-STR_TSA_USERTYPE(<i>name</i>)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()
```

```
FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *fmrstr:* protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)
```

```
/* Directory entry applies to all subsequent MEMFILE entries */
```

```
FMSTR_TSA_DIRECTORY("/text_files") /* entering a new virtual directory */
```

(continues on next page)

(continued from previous page)

```

/* The readme.txt file will be accessible at the fmstr://text_files/readme.txt URL */
FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt)) /* memory-mapped file */

/* Files can also be specified with a full path so the DIRECTORY entry does not apply */
FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index)) /* memory-mapped file */
FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) /* memory-mapped file */

/* Hyperlinks can point to a local MEMFILE object or to the Internet */
FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

/* Project file links simplify opening the projects from any URLs */
FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
FMSTR_TSA_PROJECT("Full Project (online)", "http://mycompany.com/prj/demo.pmp")

FMSTR_TSA_TABLE_END()

```

TSA API

FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR buffAddr, FMSTR_SIZE buffSize);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *buffAddr* [in] - address of the memory buffer for the dynamic TSA table
- *buffSize* [in] - size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

FMSTR_TsaAddVar

Prototype

```

FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR tsaName, FMSTR_TSATBL_STRPTR
↪ tsaType,
    FMSTR_TSATBL_VOIDPTR varAddr, FMSTR_SIZE32 varSize,
    FMSTR_SIZE flags);

```

- Declaration: *freemaster.h*

- Implementation: *freemaster_tsa.c*

Arguments

- *tsaName* [in] - name of the object
- *tsaType* [in] - name of the object type
- *varAddr* [in] - address of the object
- *varSize* [in] - size of the object
- *flags* [in] - access flags; a combination of these values:
 - *FMSTR_TSA_INFO_RO_VAR* — read-only memory-mapped object (typically a variable)
 - *FMSTR_TSA_INFO_RW_VAR* — read/write memory-mapped object
 - *FMSTR_TSA_INFO_NON_VAR* — other entry, describing structure types, structure members, enumerations, and other types

Description This function can be called only when the dynamic TSA table is enabled by the *FMSTR_USE_TSA_DYNAMIC* configuration option and when the *FMSTR_SetUpTsaBuff* function call is made to assign the dynamic TSA table memory. This function adds an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See [TSA table definition](#) for more details about the TSA table entries.

Application Commands API

FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Description This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the *FMSTR_APPCMDRESULT_NOCMD* constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the *FMSTR_AppCmdAck* call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The *FMSTR_GetAppCmd* function does not report the commands for which a callback handler function exists. If the *FMSTR_GetAppCmd* function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns *FMSTR_APPCMDRESULT_NOCMD*.

FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* dataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *dataLen* [out] - pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see [FMSTR_GetAppCmd](#)).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the FMSTR_AppCmdAck call, copy the data out to a private buffer.

FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT resultCode);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultCode* [in] - the result code which is to be returned to FreeMASTER

Description This function is used when the Application Command processing finishes in the application. The resultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(FMSTR_ADDR resultDataAddr, FMSTR_SIZE resultDataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultDataAddr* [in] - pointer to the data buffer that is to be copied to the Application Command data buffer
- *resultDataLen* [in] - length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description This function can be used before the Application Command processing finishes, when there are data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use FMSTR_GetAppCmdData and the data buffer after FMSTR_AppCmdSetResponseData is called.

Note: The current version of FreeMASTER does not support the Application Command response data.

FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(FMSTR_APPCMD_CODE appCmdCode, FMSTR_
↳ PAPPCMDFUNC callbackFunc);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *appCmdCode* [in] - the Application Command code for which the callback is to be registered
- *callbackFunc* [in] - pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return value This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when trying to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is

```
FMSTR_APPCMD_RESULT HandlerFunction(FMSTR_APPCMD_CODE nAppcmd,
FMSTR_APPCMD_PDATA pData, FMSTR_SIZE nDataLen);
```

Where:

- *nAppcmd* -Application Command code
- *pData* —points to the Application Command data received (if any)
- *nDataLen* —information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

Note: The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the FMSTR_RegisterAppCmdCall function always fails.

Pipes API

FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT pipePort, FMSTR_PPIPEFUNC pipeCallback,
    FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize,
    FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize,
    FMSTR_U8 type, const FMSTR_CHAR *name);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipePort* [in] - port number that identifies the pipe for the client
- *pipeCallback* [in] - pointer to the callback function that is called whenever a pipe data status changes
- *pipeRxBuff* [in] - address of the receive memory buffer
- *pipeRxSize* [in] - size of the receive memory buffer
- *pipeTxBuff* [in] - address of the transmit memory buffer
- *pipeTxSize* [in] - size of the transmit memory buffer
- *type* [in] - a combination of FMSTR_PIPE_MODE_XXX and FMSTR_PIPE_SIZE_XXX constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.
- *name* [in] - user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

Description This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the FMSTR_PipeRead call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the FMSTR_PipeWrite call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The *PipeHandler* name is only a placeholder and must be defined by the application.


```
void PipeHandler(FMSTR_HPIPE pipeHandle);
```

FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call

Description This function de-initializes the pipe object. No data can be received or sent on the pipe after this call.

FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,  
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data to be written
- *pipeDataLen* [in] - length of the data to be written
- *writeGranularity* [in] - size of the minimum unit of data which is to be written

Description This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the *nGranularity* value equal to the *nLength* value, all data are considered as one chunk which is either written successfully as a whole or not at all. The *nGranularity* value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE readGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data buffer to be filled with the received data
- *pipeDataLen* [in] - length of the data to be read
- *readGranularity* [in] - size of the minimum unit of data which is to be read

Description This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The readGranularity argument can be used to copy the data in larger chunks in the same way as described in the FMSTR_PipeWrite function.

API data types This section describes the data types used in the FreeMASTER driver. The information provided here can be useful when modifying or porting the FreeMASTER Communication Driver to new NXP platforms.

Note: The licensing conditions prohibit use of FreeMASTER and the FreeMASTER Communication Driver with non-NXP MPU or MCU products.

Public common types The table below describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Type name	Description
<i>FM-STR_ADDR</i> For example, this type is defined as long integer on the 56F8xxx platform where the 24-bit addresses must be supported, but the C-pointer may be only 16 bits wide in some compiler configurations.	Data type used to hold the memory address. On most platforms, this is normally a C-pointer, but it may also be a pure integer type.
<i>FM-STR_SIZE</i> It is required that this type is unsigned and at least 16 bits wide integer.	Data type used to hold the memory block size.
<i>FM-STR_BOOL</i> This type is used only in zero/non-zero conditions in the driver code.	Data type used as a general boolean type.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command code.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to create the Application Command data buffer.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command result code.

Public TSA types The table describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included in the user application indirectly by the *freemaster.h* file.

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

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

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

Internal types The table describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and they are not available in the application code.

<i>FMSTR_U8</i>	The smallest memory entity.
On the vast majority of platforms, this is an unsigned 8-bit integer.	
On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.	
<i>FMSTR_U16</i>	Unsigned 16-bit integer.
<i>FMSTR_U32</i>	Unsigned 32-bit integer.
<i>FMSTR_S8</i>	Signed 8-bit integer.
<i>FMSTR_S16</i>	Signed 16-bit integer.
<i>FMSTR_S32</i>	Signed 32-bit integer.
<i>FMSTR_FLOAT</i>	4-byte standard IEEE floating-point type.
<i>FMSTR_FLAGS</i>	Data type forming a union with a structure of flag bit-fields.
<i>FMSTR_SIZE8</i>	Data type holding a general size value, at least 8 bits wide.
<i>FMSTR_INDEX</i>	General for-loop index. Must be signed, at least 16 bits wide.
<i>FMSTR_BCHR</i>	A single character in the communication buffer.
Typically, this is an 8-bit unsigned integer, except for the DSP platforms where it is a 16-bit integer.	
<i>FMSTR_BPTR</i>	A pointer to the communication buffer (an array of <i>FMSTR_BCHR</i>).

Document references

Links

- This document online: <https://mcuxpresso.nxp.com/mcuxsdk/latest/html/middleware/freemaster/doc/index.html>

- FreeMASTER tool home: www.nxp.com/freemaster
- FreeMASTER community area: community.nxp.com/community/freemaster
- FreeMASTER GitHub code repo: <https://github.com/nxp-mcuxpresso/mcux-freemaster>
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

- *FreeMASTER Usage Serial Driver Implementation* (document [AN4752](#))
- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document [AN4771](#))
- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document [AN4860](#))

Revision history This Table summarizes the changes done to this document since the initial release.

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support.
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. added MCAN interface. Folders structure at the installation destination was rearranged.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.
4.3	10/2024	Added description of Network and Segger J-Link RTT interface configuration. Accompanying the MCUXpresso SDK version 24.12.00.
4.4	04/2025	Added Zephyr-specific information. Accompanying the MCUXpresso SDK version 25.06.00.

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

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

[Readme](#)

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme