



MCUXpresso SDK Documentation

Release 25.09.00-pvw2



NXP
Aug 12, 2025



Table of contents

1	EVK-MIMX8ULP	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	MCUXpresso SDK Release Notes for EVK-MIMX8ULP	3
1.3	Getting Started with MCUXpresso SDK GitHub	22
1.3.1	Getting Started with MCUXpresso SDK Repository	23
1.4	Release Notes	35
1.4.1	MCUXpresso SDK Release Notes	35
1.5	ChangeLog	40
1.5.1	MCUXpresso SDK Changelog	40
1.6	Driver API Reference Manual	40
1.7	Middleware Documentation	41
1.7.1	Multicore	41
1.7.2	FreeMASTER	41
1.7.3	FreeRTOS	41
2	MIMX8UD7	43
2.1	ACMP: Analog Comparator Driver	43
2.2	Battery-Backed Non-Secure Module	51
2.3	CACHE: CACHE Memory Controller	53
2.4	CASPER: The Cryptographic Accelerator and Signal Processing Engine with RAM sharing	56
2.5	casper_driver	56
2.6	casper_driver_pkha	60
2.7	Clock	63
2.8	CMC: Core Mode Controller Driver	119
2.9	MIPI CSI2 RX: MIPI CSI2 RX Driver	131
2.10	DAC12: 12-bit Digital-to-Analog Converter Driver	137
2.11	EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	143
2.12	ENET: Ethernet MAC Driver	178
2.13	EPDC: Electrophoretic Display Controller	209
2.14	EWM: External Watchdog Monitor Driver	228
2.15	FGPIO Driver	231
2.16	FlexCAN: Flex Controller Area Network Driver	231
2.17	FlexCAN Driver	231
2.18	FlexCAN eDMA Driver	278
2.19	FlexIO: FlexIO Driver	280
2.20	FlexIO Driver	280
2.21	FlexIO eDMA SPI Driver	298
2.22	FlexIO eDMA UART Driver	301
2.23	FlexIO I2C Master Driver	304
2.24	FlexIO I2S Driver	312
2.25	FlexIO SPI Driver	323
2.26	FlexIO UART Driver	336
2.27	FLEXSPI: Flexible Serial Peripheral Interface Driver	346
2.28	FLEXSPI eDMA Driver	363

2.29	I3C: I3C Driver	366
2.30	I3C Common Driver	368
2.31	I3C Master Driver	370
2.32	I3C Slave Driver	396
2.33	Iomuxc_driver	409
2.34	ISI: Image Sensing Interface	458
2.35	Common Driver	474
2.36	LCDIF: LCD interface	485
2.37	Lin_lpuart_driver	498
2.38	LPADC: 12-bit SAR Analog-to-Digital Converter Driver	506
2.39	LPI2C: Low Power Inter-Integrated Circuit Driver	524
2.40	LPI2C Master Driver	525
2.41	LPI2C Master DMA Driver	540
2.42	LPI2C Slave Driver	542
2.43	LPIT: Low-Power Interrupt Timer	552
2.44	LPSPi: Low Power Serial Peripheral Interface	559
2.45	LPSPi Peripheral driver	559
2.46	LPSPi eDMA Driver	581
2.47	LPTMR: Low-Power Timer	588
2.48	LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver	593
2.49	LPUART Driver	593
2.50	LPUART eDMA Driver	612
2.51	MCM: Miscellaneous Control Module	615
2.52	MIPI DSI Driver	619
2.53	MIPI_DSI: MIPI DSI Host Controller	637
2.54	MRT: Multi-Rate Timer	637
2.55	MU: Messaging Unit	642
2.56	PDM: Microphone Interface	660
2.57	PDM Driver	660
2.58	PDM EDMA Driver	678
2.59	POWERQUAD: PowerQuad hardware accelerator	682
2.60	PXP: Pixel Pipeline	711
2.61	Reset	747
2.62	RGPIO: Rapid General-Purpose Input/Output Driver	750
2.63	RGPIO Driver	753
2.64	RTD_CMC: Real Time Domain Core Mode Controller Driver	756
2.65	SAI: Serial Audio Interface	766
2.66	SAI Driver	766
2.67	SAI EDMA Driver	791
2.68	SEMA42: Hardware Semaphores Driver	798
2.69	Sentinel	802
2.70	SFA: Signal Frequency Analyser	808
2.71	SIM: System Integration Module Driver	817
2.72	Soc_mipi_dsi	818
2.73	SPDIF: Sony/Philips Digital Interface	818
2.74	SPDIF eDMA Driver	832
2.75	SYSPM: System Performance Monitor	835
2.76	TPM: Timer PWM Module	838
2.77	TRDC: Trusted Resource Domain Controller	854
2.78	TRGMUX: Trigger Mux Driver	875
2.79	TSTMR: Timestamp Timer Driver	876
2.80	Upower	877
2.81	USDHC: Ultra Secured Digital Host Controller Driver	887
2.82	WDOG32: 32-bit Watchdog Timer	917
2.83	WUU: Wakeup Unit driver	923
2.84	XRDC: Extended Resource Domain Controller	928

3 Middleware

947

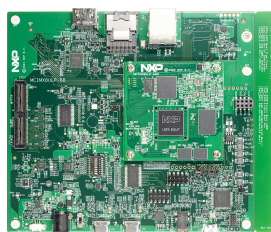
3.1	Motor Control	947
3.1.1	FreeMASTER	947
3.2	MultiCore	984
3.2.1	Multicore SDK	984
4	RTOS	1081
4.1	FreeRTOS	1081
4.1.1	FreeRTOS kernel	1081
4.1.2	FreeRTOS drivers	1087
4.1.3	backoffalgorithm	1087
4.1.4	corehttp	1090
4.1.5	corejson	1092
4.1.6	coremqtt	1095
4.1.7	coremqtt-agent	1098
4.1.8	corepkcs11	1102
4.1.9	freertos-plus-tcp	1105

This documentation contains information specific to the evkmimx8ulp board.

Chapter 1

EVK-MIMX8ULP

1.1 Overview



MCU device and part on board is shown below:

- Device: MIMX8UD7
- PartNumber: MIMX8UD7DVP08

1.2 Getting Started with MCUXpresso SDK Package

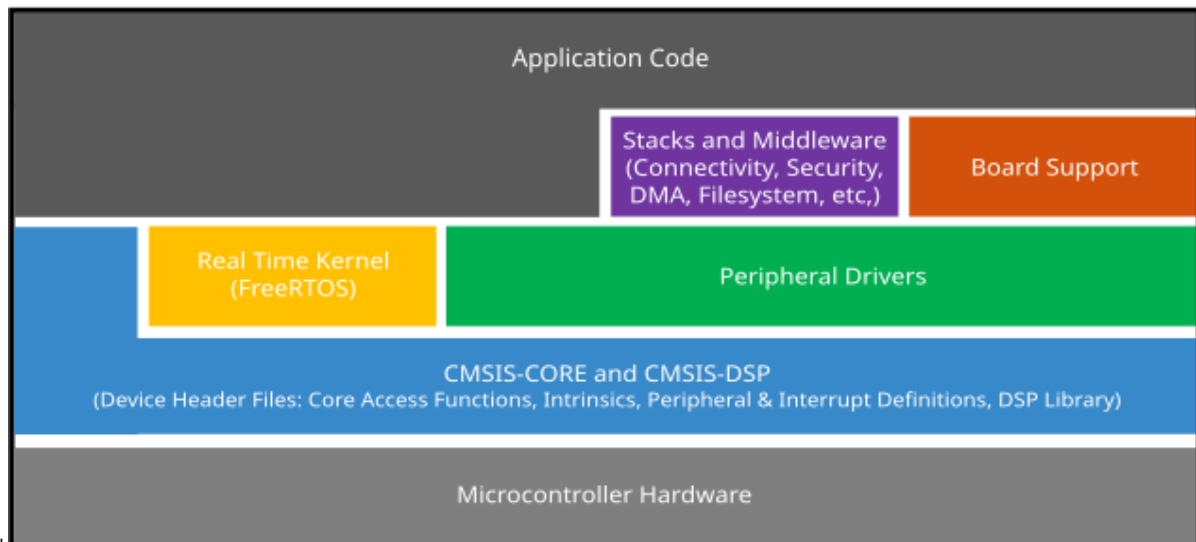
1.2.1 MCUXpresso SDK Release Notes for EVK-MIMX8ULP

Overview

The MCUXpresso Software Development Kit (MCUXpresso SDK) provides comprehensive software source code to be executed in the i.MX 8ULP M33 core. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. These drivers can be used standalone or collaboratively with the A35 cores running another Operating System (such as Linux OS Kernel). Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to demo applications. The MCUXpresso SDK also contains RTOS kernels, device stack, and various other middleware to support rapid development.

For supported toolchain versions, see the *MCUXpresso SDK Release Notes for EVK-MIMX8ULP* (document MCUXSDKIMX8ULPRN).

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



MCUXpresso SDK board support folders

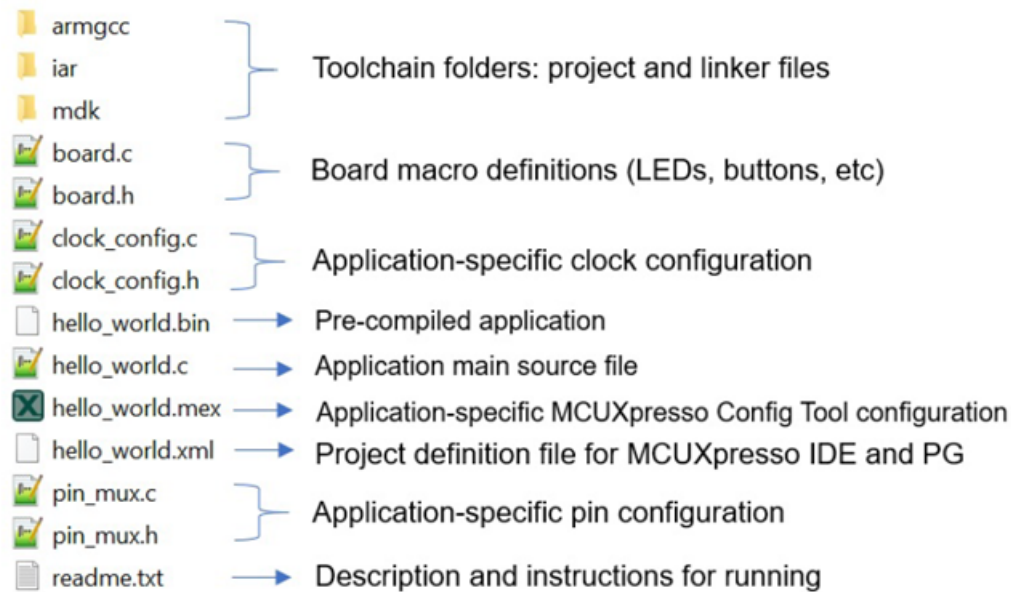
MCUXpresso SDK provides example applications for development and evaluation boards. Board support packages are found inside the top level `<board_name>` folder, and each supported board has its own folder (an MCUXpresso SDK package can support multiple boards). Within each `<board_name>` folder, there are various sub-folders for each example they contain. These include (but are not limited to):

- `demo_apps`: Applications intended to 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 intended to concisely illustrate 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 are used.
- `rtos_examples`: Basic FreeRTOS examples showcasing the use of various RTOS objects (semaphores, queues, and so on) and interfacing with the MCUXpresso SDK's RTOS drivers.
- `cmsis_driver_examples`: Simple applications intended to concisely illustrate how to use CMSIS drivers.
- `multicore_examples`: Simple applications intended to concisely illustrate how to use middleware/multicore stack.
- `mmcau_examples`: Simple applications intended to concisely illustrate how to use middleware/mmcau stack.

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

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

The following figure shows the contents of the `hello_world` application folder.



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 folders](#)

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

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

- `devices/<device_name>`: 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. RTOS files are in the `rtos` folder. The core files of each of these projects are shared, so modifying one could have potential impacts on other projects that depend on that file.

Note: The RPMsg-Lite library is located in the `<install_dir>/middleware/multicore/rpmsg-lite` folder. For detailed information about the RPMsg-Lite library, see the *RPMsg-Lite User's Guide*, open the `index.html` located in the `<install_dir>/middleware/multicore/rpmsg_lite/doc` folder.

Note: The package does not include Xplorer IDE and DSP Fusion user guide. If you want to run examples related to DSP Fusion, contact the NXP representative (FAE/SE).

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

Toolchain introduction

The MCUXpresso SDK release for i.MX 8ULP includes the build system to be used with some toolchains. This chapter lists and explains the supported toolchains.

Compiler/Debugger The release supports building and debugging with the toolchains listed in Table 1.

You can choose the appropriate one for development.

- Arm GCC + SEGGER J-Link GDB Server. This is a command-line tool option and it supports both Windows OS and Linux OS.
- IAR Embedded Workbench for Arm and SEGGER J-Link software. The IAR Embedded Workbench is an IDE integrated with editor, compiler, debugger, and other components. The SEGGER J-Link software provides the driver for the J-Link Plus debug probe and supports the device to attach, debug, and download.

Compiler/Debugger		Supported host OS	Debug probe	Tool website
Arm GDB Server	GCC/J-Link	Windows OS/Linux OS	J-Link Plus	developer.arm.com/open-source/gnu-toolchain/gnu-rm

www.segger.com

| | IAR/J-Link | Windows OS | J-Link Plus | www.iar.com

www.segger.com

|

Download the corresponding tools for the specific host OS from the website.

Note: To support i.MX 8ULP, the patch for IAR and SEGGER J-Link should be installed. The patch named [iar_segger_support_patch_imx8ulp.zip](#) can be used with the MCUXpresso SDK. See [readme.txt](#) in the patch for additional information about patch installation.

Parent topic: [Toolchain introduction](#)

Running a Demo Application Using Arm GCC

This section describes the steps to configure the command-line Arm GCC tools to build, run, and debug demo applications. This section also lists the necessary driver libraries provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the MIMX8ULP hardware platform is used as an example, though these steps can be applied to any board, demo, or example application in the MCUXpresso SDK.

Note: Run an application using `imx-mkimage`. Generate and download `flash.bin` to emmc or flexspi nor flash when `DBD_EN` (Deny By Default) is fused.

Linux OS host The following sections provide steps to run a demo compiled with Arm GCC on Linux host.

Set up toolchain This section contains the steps to install the necessary components required to build and run a MCUXpresso SDK demo application with the Arm GCC toolchain, as supported by the MCUXpresso SDK.

Install GCC Arm embedded toolchain Download and run the installer from the [GNU Arm Embedded Toolchain Downloads](#) page. The GNU Arm embedded toolchain contains the GCC compiler, libraries, and other tools required for bare-metal software development. The GCC toolchain should correspond to the latest supported version, as described in the *MCUXpresso SDK Release Notes for EVK-MIMX8ULP* (document MCUXSDKIMX8ULPRN).

Note: See [How to set up Windows/Linux host system](#) for setting up Linux host before compiling the application.

Parent topic:Set up toolchain

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

```
$ export ARMGCC_DIR=<path_to_GNUARM_GCC_installation_dir>
```

Parent topic:Set up toolchain

Parent topic:Linux OS host

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

1. 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>/boards/evkmimx8ulp/demo_apps/hello_world/armgcc

2. Run the build_debug.sh script at the command-line to perform the build. The output is shown as below:

```
$ ./build_debug.sh
-- TOOLCHAIN_DIR:
-- BUILD_TYPE: debug
-- TOOLCHAIN_DIR:
-- BUILD_TYPE: debug
-- The ASM compiler identification is GNU
-- Found assembler:
-- Configuring done
-- Generating done
-- Build files have been written to:
Scanning dependencies of target hello_world.elf
< -- skipping lines -- >
[100%] Linking C executable debug/hello_world.elf
[100%] Built target hello_world.elf
```

Note: To run the application, see the [Run an application using imx-mkimage](#).

Parent topic:Linux OS host

Parent topic:[Running a Demo Application Using Arm GCC](#)

Windows OS host The following sections provide steps to run a demo compiled with Arm GCC on Windows OS host.

Set up toolchain This section contains the steps to install the necessary components required to build and run a MCUXpresso SDK demo application with the Arm GCC toolchain on Windows OS, as supported by the MCUXpresso SDK.

Install GCC Arm embedded toolchain Download and run the installer from the [GNU Arm Embedded Toolchain Downloads](#) page. The GNU Arm embedded toolchain contains the GCC compiler, libraries, and other tools required for bare-metal software development. The GCC toolchain should correspond to the latest supported version, as described in *MCUXpresso SDK Release Notes for EVK-MIMX8ULP* (document MCUXSDKIMX8ULPRN).

Note: See [How to set up Windows/Linux host system](#) for setting up Windows host before compiling the application.

Parent topic:Set up toolchain

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

```
C:\Program Files (x86)\GNU Arm Embedded Toolchain\9 2020-q2-update
```

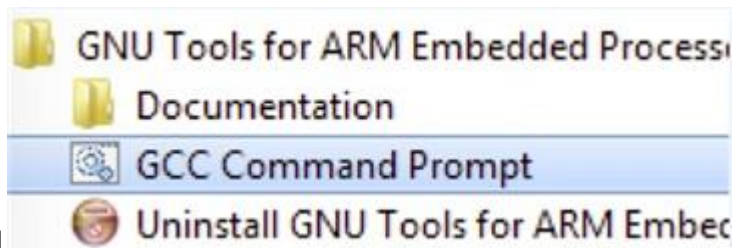
Reference the installation folder of the GNU Arm GCC embedded tools for the exact pathname.

Parent topic:Set up toolchain

Parent topic:Windows OS host

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

1. Open the GCC Arm embedded toolchain command window. To launch the window on the Windows operating system, select **Start** -> **Programs** -> **GNU Tools ARM Embedded <version>** -> **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>/boards/evkmimx8ulp/demo_apps/hello_world/armgcc
```

3. Type `build_debug.bat` at the command-line or double-click the `build_debug.bat` file in Windows Explorer to perform the build. The output is as shown in Figure 2.

```
[100%] Linking C executable debug\hello_world.elf
Memory region      Used Size  Region Size  %age Used
  m_interrupts:      768 B      768 B      100.00%
    m_text:        24704 B    253184 B      9.76%
    m_data:         2608 B     224 KB      1.14%
m_m33_suspend_ram:    0 GB      16 KB      0.00%
m_a35_suspend_ram:    0 GB      16 KB      0.00%
[100%] Built target hello_world.elf
C:\Users\... \src\mcu-sdk-2.0\boards\evkmimx8ulp\demo_apps\hello_world\armgcc>
```


|

Note: To run the application, see the [Run an application using imx-mkimage](#).

Parent topic: Windows OS host

Parent topic: [Running a Demo Application Using Arm GCC](#)

Running a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK using IAR. The `hello_world` demo application targeted for the MIMX8ULP hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

Note:

- Newer versions of the IAR are compatible with older versions of the project format. However, using an older version of the IAR to load the SDK project that uses the newer format generates an error. To use the SDK, it is recommended to upgrade the IAR version to 9.30.1.
- Run an application using `imx-mkimage`. Generate and download `flash.bin` to `emmc` or `flexspi nor flash` when `DBD_EN` (Deny By Default) is fused.

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

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

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

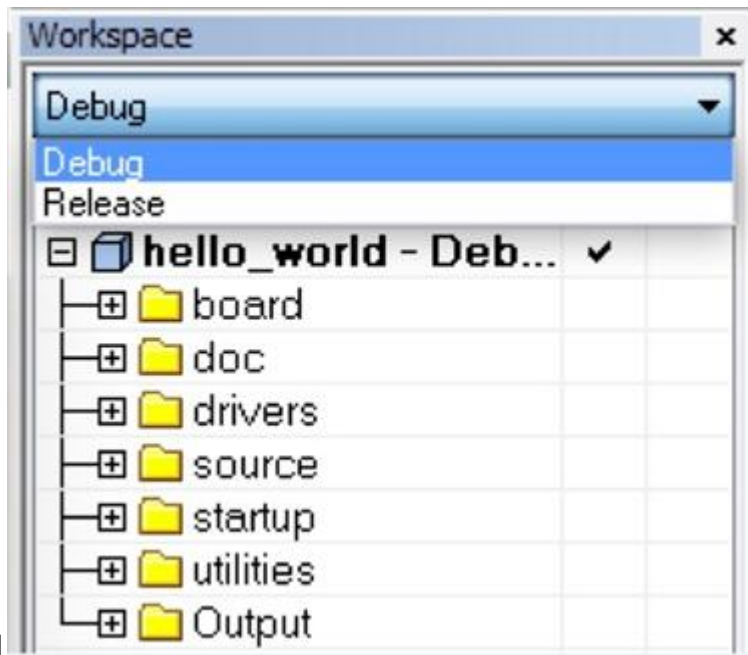
For using MIMX8ULP-EVK hardware platform as an example, the `hello_world` workspace is located at:

```
<install_dir>/boards/evkmimx8ulp/demo_apps/hello_world/iar/hello_world.eww
```

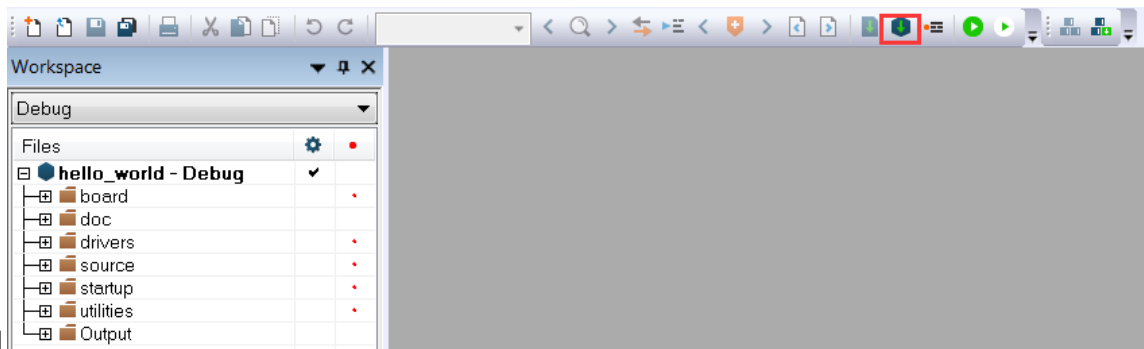
Other example applications may have additional folders in the respective paths.

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

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



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



4. The build completes without errors.

Note: To run the application, see the [Run an application using imx-mkimage](#).

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

Run an application using imx-mkimage

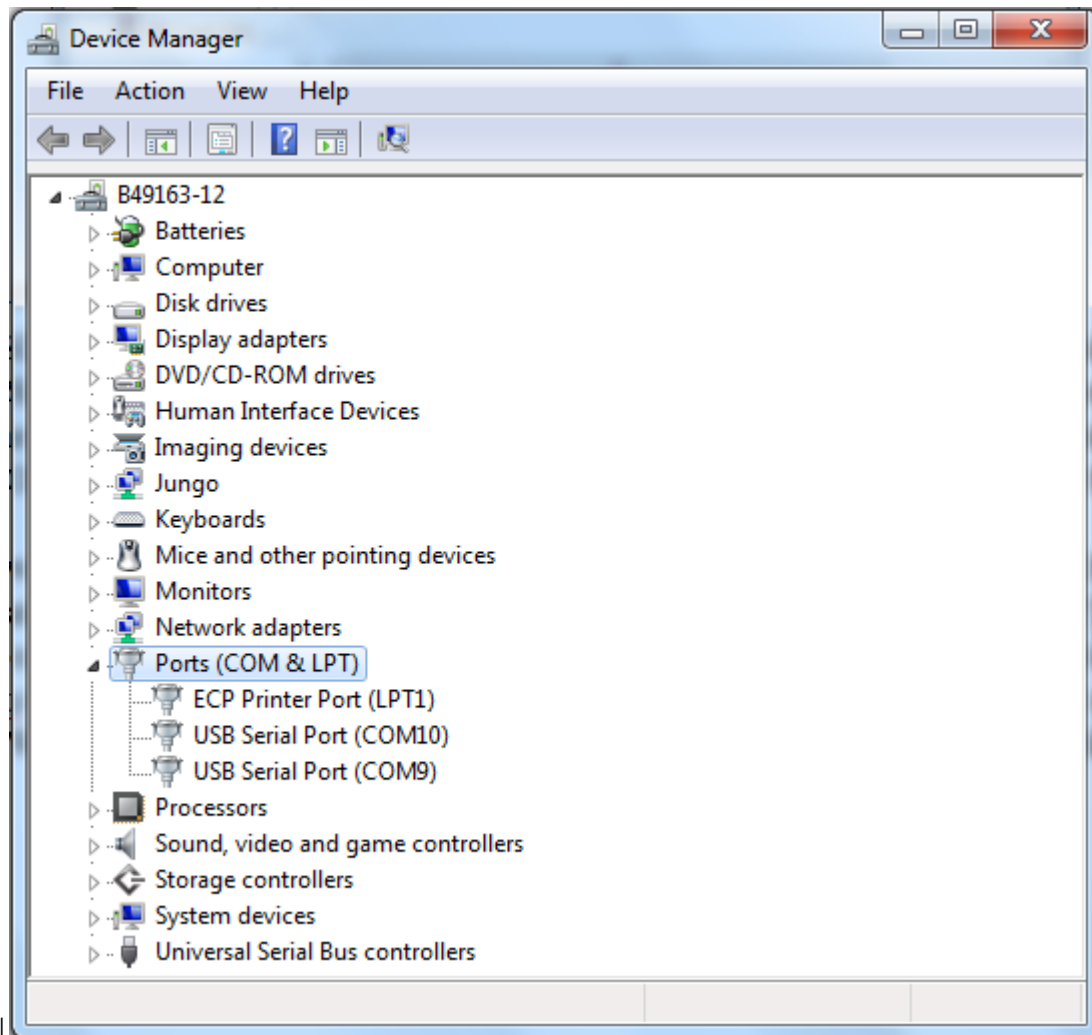
This section describes the steps to write a bootable SDK image to the eMMC/FlexSPI NOR flash for the i.MX processor.

Note: Attach core to debug code with J-Link probe.

The following steps describe how to write container image (flash.bin):

1. Connect the DEBUG UART slot on the board to your PC through the USB cable. The Windows OS installs the USB driver automatically and the Ubuntu OS finds the serial devices as well.
2. On Windows OS, open the device manager, find **USB serial Port** in **Ports (COM and LPT)**. Assume that the ports are COM9 and COM10. One port is for the debug message from the Cortex-A35 and the other is for the Cortex-M33. The port number is allocated randomly, so opening both is beneficial for development. On Ubuntu OS, find the TTY device with

name `/dev/ttyUSB*` to determine your debug port. Similar to Windows OS, opening both is beneficial for development.



3. Generate m33 firmware:

- **For RAM target:**

```
$ ./build_debug.sh
```

or

```
$ ./build_release.sh
```

- **For FLASH target(XIP):**

```
$ ./build_flash_debug.sh
```

or

```
$ ./build_flash_release.sh
```

4. Get imx-mkimage, s400 firmware(mx8sulp2-ahab-container.img), OPTEE(tee.bin), upower firmware(upower.bin), uboot-spl(u-boot-spl.bin), uboot(u-boot.bin), and TF-A(bl31.bin) from the Linux release package.

1. Clone the imx-mkimage from NXP public git.

```
$ git clone https://github.com/nxp-imx/imx-mkimage
```

2. Check out the correct branch. The branch name is named after Linux release version which is compatible with the SDK. You can get the version information from corresponding Linux Release Notes document.

```
$ cd imx-mkimage
$ git checkout [branch name]
```

3. Get s400 firmware(mx8ulpa2-ahab-container.img).

```
$ cp mx8ulpa2-ahab-container.img iMX8ULP/
```

4. Get upower firmware(upower.bin).

```
$ cp upower.bin iMX8ULP/
```

5. Get u-boot-spl.bin and u-boot.bin.

- For EVK-MIMX8ULP:

```
$ cp u-boot-spl.bin-imx8ulpevk-sd iMX8ULP/u-boot-spl.bin
$ cp u-boot-imx8ulpevk.bin-sd iMX8ULP/u-boot.bin
```

- For EVK9-MIMX8ULP:

```
$ cp u-boot-spl.bin-imx8ulp-9x9-lpddr4-evk-sd iMX8ULP/u-boot-spl.bin
$ cp u-boot-imx8ulp-9x9-lpddr4-evk.bin-sd iMX8ULP/u-boot.bin
```

6. Get bl31.bin.

```
$ cp bl31-imx8ulp.bin-optee iMX8ULP/bl31.bin
```

5. Generate container image table with imx-mkimage:

boot type	A35	M33	SW5[8:1]
Single Boot	make SOC=iMX8ULP flash_singleboot For RAM target:		

make SOC=iMX8ULP flash_singleboot_m33 **Note:** Does not support pack Flash target into flash.bin when boot type is single boot type.

| 1000_xx00 Single Boot-eMMC | | make SOC=iMX8ULP flash_singleboot_flexspi For RAM target:

make SOC=iMX8ULP flash_singleboot_m33_flexspi **Note:** Does not support pack Flash target into flash.bin when boot type is single boot type.

| 1010_xx00 Single Boot-Nor | | Dual Boot | make SOC=iMX8ULP flash_dualboot | For RAM target:
make SOC=iMX8ULP flash_dualboot_m33

For Flash target: make SOC=iMX8ULP flash_dualboot_m33_xip

| 1000_0010 A35-eMMC/M33-Nor | | make SOC=iMX8ULP flash_dualboot_flexspi | 1010_0010
A35-Nor/M33-Nor | | Low Power Boot | make SOC=iMX8ULP flash_dualboot | 1000_00x1 A35-
eMMC/M33-Nor | | make SOC=iMX8ULP flash_dualboot_flexspi | 1010_00x1 A35-Nor/M33-Nor |

****Note:****

- For details, see `imx-mkimage/iMX8ULP/README`.
- Does not support pack Flash target firmware to flash.bin when boot type is single boot type.

(continues on next page)

(continued from previous page)

- RAM target: debug/release.
- Flash target: `flash\_debug/flash\_release`.
- Need generate two flash.bin and download to emmc/flexspi2 nor flash of a35 and flexspi0 nor flash of m33,  one for A35, another one for M33 when boot type is dual boot type or low power boot type.

6. Build the application (for example, hello_world), get binary image sdk20-app.bin, copy to imx-mkimage project folder iMX8ULP/ and rename to m33_image.bin.

```
cp sdk20-app.bin <imx-mkimage path>/iMX8ULP/m33_image.bin
```

7. Under imx-mkimage project folder, execute the following command to generate m33 container image.

1. When boot type is dual boot/low power boot type:

For RAM (TCM) target:

```
make SOC=iMX8ULP flash_dualboot_m33 (write flash.bin to flexspi0 nor flash of m33;
```

For Flash target:

```
make SOC=iMX8ULP flash_dualboot_m33_xip (write flash.bin to flexspi0 nor flash of m33);
```

2. When boot type is single boot type:

```
for RAM (TCM) target and sw5[8:1] = 1000_xx00 Single Boot-eMMC:
make SOC=iMX8ULP flash_singleboot_m33 (write flash.bin to emmc);
```

```
for RAM (TCM) target and sw5[8:1] = 1010_xx00 Single Boot-Nor:
make SOC=iMX8ULP flash_singleboot_m33_flexspi (write flash.bin to flexspi2 nor flash of
a35);
```

8. Copy the flash.bin image to your tftpboot server.

9. Write flash.bin to flexspi0 nor flash. There are two ways:

1. Write flash.bin to flexspi0 nor flash with uboot.

1. Switch to single boot type (sw[8:1]=1000 0000) and boot the board, assuming your board can boot to U-Boot.
2. At the U-Boot console, execute following commands to download image (from network) and flash to FlexSPI0 NOR flash.

```
setenv serverip <tftpboot server ip>
dhcp
tftpboot 0xa0000000 flash.bin
setenv erase_unit 1000
setexpr erase_size ${filesize} + ${erase_unit}
setexpr erase_size ${erase_size} / ${erase_unit}
setexpr erase_size ${erase_size} * ${erase_unit}
sf probe 0:0
sf erase ${erase_size}
sf write 0xa0000000 0 ${filesize}
```

2. Write flash.bin to flexspi0 nor flash with JLink:

```
J-Link>connect
Device>
TIF>s (Choose target interface as SWD, unless failed to do anything)
Speed>
```

(continues on next page)

(continued from previous page)

```
J-Link>r
J-Link>h
J-Link>loadbin flash.bin 0x4000000
```

10. Write flash.bin to emmc with uuu (only for the RAM target):

1. Start uuu.

```
uuu -b emmc workable-flash.bin flash.bin (workable-flash.bin: uboot and m33 image are workable)
```

2. Enter serial download mode.

1. Change SW5[8:1] to 01xx_xxxx Serial Downloader.
2. Enter serial download mode with uboot.

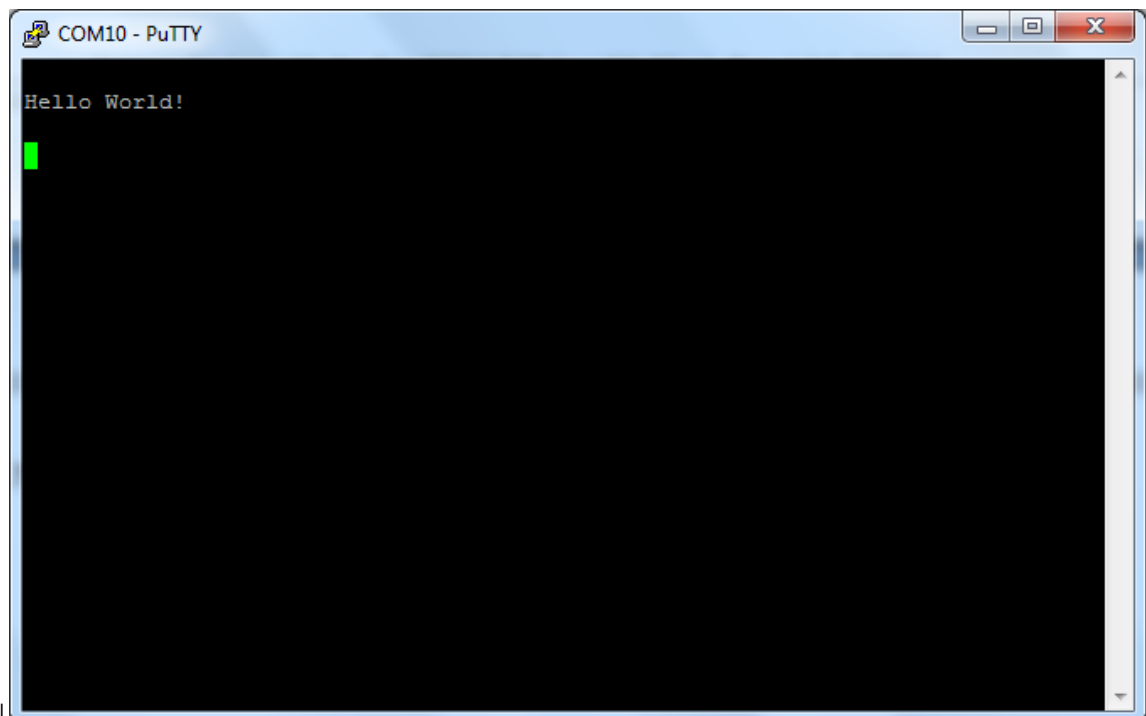
```
=> fastboot 0
```

11. Open another terminal application on the PC, such as PuTTY 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:

- 115200
- No parity
- 8 data bits
- 1 stop bit

12. Power off and switch to low-power boot mode (sw5[8:1]=1000 0001), then repower the board.

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



Memory attribution map after doing handshake

The memory attribution map settings after the handshake procedure is successful between Cortex-M33 and Cortex-A35.

Name	Memory block checker/ Memory region checker (MBC/MRC)	Resulting access level
FLEXSPI1 (alias) 0x5000_0000	Non Secure	Non Secure
FLEXSPI1 0x4000_0000	Non Secure	Non Secure
PBridge1 FlexCAN0 (alias) 0x380A_8000	Non Secure	Non Secure
PBridge1 SAI0 (alias) 0x3809_C000	Non Secure	Non Secure
PBridge1 LPUART1 (alias) 0x3809_B000	Non Secure	Non Secure
PBridge1 LPI2C0 (alias) 0x3809_8000	Non Secure	Non Secure
PBridge1 FlexSPI1 (alias) 0x3809_2000	Non Secure	Non Secure
PBridge0 LPSP11 (alias) 0x3803_F000	Non Secure	Non Secure
PBridge0 FlexIO0 (alias) 0x3803_C000	Non Secure	Non Secure
PBridge0 FlexSPI0 (alias) 0x3803_9000	Non Secure	Non Secure
PBridge1 FlexCAN0 0x280A_8000	Non Secure	Non Secure
PBridge1 SAI0 0x2809_C000	Non Secure	Non Secure
PBridge1 LPUART1 0x2809_B000	Non Secure	Non Secure
PBridge1 LPI2C0 0x2809_8000	Non Secure	Non Secure
PBridge1 FlexSPI1 0x2809_2000	Non Secure	Non Secure
PBridge0 LPSP11 0x2803_F000	Non Secure	Non Secure
PBridge0 FlexIO0 0x2803_C000	Non Secure	Non Secure
PBridge0 FlexSPI0 0x2803_9000	Non Secure	Non Secure
SSRAM P6 (alias) 0x3006_0000	Non Secure	Non Secure
SSRAM P5 (alias) 0x3004_0000	Non Secure	Non Secure
SSRAM P4 (alias) 0x3003_0000	Non Secure	Non Secure
SSRAM P3 (alias) 0x3002_0000	Non Secure	Non Secure
SSRAM P2 (alias) 0x3001_0000	Non Secure	Non Secure
SSRAM P1 (alias) 0x3000_8000	Non Secure	Non Secure
SSRAM P0 (alias) 0x3000_0000	Non Secure	Non Secure

contin

Table 1 – continued from previous page

Name	Memory block checker/ Memory region checker (MBC/MRC)	Resulting access level
SSRAM P6 0x2006_0000	Non Secure	Non Secure
SSRAM P5 0x2004_0000	Non Secure	Non Secure
SSRAM P4 0x2003_0000	Non Secure	Non Secure
SSRAM P3 0x2002_0000	Non Secure	Non Secure
SSRAM P2 0x2001_0000	Non Secure	Non Secure
SSRAM P1 0x2000_8000	Non Secure	Non Secure
SSRAM P0 0x2000_0000	Non Secure	Non Secure
SSRAM P7 (alias) 0x1FFC_0000	Non Secure	Non Secure
FlexSPI0 (alias) 0x1400_0000	Non Secure	Non Secure
SSRAM P7 0x0FFC_0000	Non Secure	Non Secure
FlexSPI0 0x0400_0000	Non Secure	Non Secure

Note:

1. Assign Domain 1 for DMA1, USB0, USB1, ENET, USDHC0, USDHC1, USDHC2, and CAAM Master.
2. The bus attribute for DMA1, USB0, USB1, ENET, USDHC0, USDHC1, and USDHC2 is Non Secure.
3. The bus attribute for CAAM Master is Secure.
4. Security level of MBC/MRC settings of other memory space that are not be shown in the table for Domain 1 are Secure, so master cannot access resources that are controlled by MBC/MRC in other memory spaces when master is in Non Secure state.

Name	MBC/MRC	Resulting access level	— :— :————— ::	PBridge1	Non Secure
	Non Secure	0x280F_FFFF	0x2808_0000	SSRAM P2	Secure
	No Access	0x2001FFFF	0x20018000	Non Secure	Non Secure
	0x20017FFF	0x20010000			

Note:

1. Assign Domain 1 for DMA1, USB0, USB1, ENET, UDSHC0, UDSHC1, UDSHC2, and CAAM Master.
2. Security level of MBC/MRC settings of other memory space that are not shown in the table for Domain 1 are Secure, so the master cannot access resources that are controlled by MBC/MRC in other memory spaces.

Name	SAU	IDAU	MBC/MRC	Resulting access level					:	
			GPIOC_REGS	(alias)	Secure	Secure	Secure	Secure	0x3882_FFFF	
0x3882_0000			GPIOB_REGS	(alias)	Secure	Secure	Secure	Secure	0x3881_FFFF	
0x3881_0000			GPIOA_REGS	(alias)	Secure	Secure	Secure	Secure	0x3880_FFFF	
0x3880_0000			MICFIL	(alias)	Secure	Secure	Secure	Secure	0x3811_10AB	
0x3811_1000			SAI3 (alias)	Secure	Secure	Secure	Secure	0x3811_00FF		0x3811_0000
SAI2	(alias)		Secure	Secure	Secure	Secure	0x3810_F0FF		0x3810_F000	LP-
SPI3	(alias)		Secure	Secure	Secure	Secure	0x3810_E7FF		0x3810_E000	LP-
SPI2	(alias)		Secure	Secure	Secure	Secure	0x3810_D7FF			0x3810_D000
LPUART3	(alias)		Secure	Secure	Secure	Secure	0x3810_C02F			0x3810_C000

LPUART2	(alias)	Secure	Secure	Secure	Secure	0x3810_B02F		0x3810_B000	
I3C1	(alias)	Secure	Secure	Secure	Secure	0x3810_AFFF		0x3810_A000	
LPI2C3	(alias)	Secure	Secure	Secure	Secure	0x3810_9173		0x3810_9000	
LPI2C2	(alias)	Secure	Secure	Secure	Secure	0x3810_8173		0x3810_8000	
MRT	(alias)	Secure	Secure	Secure	Secure	0x3810_70FF		0x3810_7000	
(alias)	Secure	Secure	Secure	Secure	0x3810_6087		0x3810_6000		TPM3
(alias)	Secure	Secure	Secure	Secure	0x3810_5087		0x3810_5000		TPM2
(alias)	Secure	Secure	Secure	Secure	0x3810_2047		0x3810_2000		PCC2
(alias)	Secure	Secure	Secure	Secure	0x3810_100F		0x3810_1000		WDOG2
(alias)	Secure	Secure	Secure	Secure	0x3810_028F		0x3810_0000		MU1_B
(alias)	Secure	Secure	Secure	Secure	0x380A_BFFF		0x380A_8000		FlexCAN0
(alias)	Secure	Secure	Secure	Secure	0x380A_2303		0x380A_2000		ADC1
(alias)	Secure	Secure	Secure	Secure	0x380A_1AEB		0x380A_1000		IOMUXC0
(alias)	Secure	Secure	Secure	Secure	0x3809_D0FF		0x3809_D000		SAI1

Note:

1. SAU is disabled.
2. Cortex-M33 can access all of the secure resources. All of the resources (the security level of these resources that are controlled by MBC/MRC) are secure when Cortex-M33 is in secure state.
3. Assign domain 6 for Cortex-M33.

Name	MBC/MRC	Se-	Resulting level	access
PBridge1 IOMUXC0 0x280A_1000	Non cure	Se-	Non Secure	0x280A_1AEB
PBridge1 LPI2C0 0x2809_8000	Non cure	Se-	Non Secure	0x2809_8173
PBridge1 TPM0 0x2809_5000	Non cure	Se-	Non Secure	0x2809_5087
PBridge1 PCC1 0x2809_1000	Non cure	Se-	Non Secure	0x2809_10BF
PBridge0 FlexSPI0 0x2803_9000	Non cure	Se-	Non Secure	0x2803_92FF
PBridge0 SEMA42_0 0x2803_7000	Non cure	Se-	Non Secure	0x2803_7043
PBridge0 CGC0 0x2802_F000	Non cure	Se-	Non Secure	0x2802_FFFF
PBridge0 SIM0-S 0x2802_B000	Non cure	Se-	Non Secure	0x2802_B3FF
S400 MU-AP of EdgeLock secure enclave 0x2702_0000	Non cure	Se-	Non Secure	0x2702_028C
FSB of EdgeLock secure enclave 0x2701_0000	Non cure	Se-	Non Secure	0x2701_0BFC
SSRAM P7 0x1FFF_8000	Non cure	Se-	Non Secure	0x1FFF_FFFF
Secure 0x1FFC_0000	Secure		0x1FFF_7FFF	
FlexSPI0 0x0400_0000	Non cure	Se-	Non Secure	0x0BFF_FFFF

Note:

1. Security level of MBC/MRC settings of Other memory space that are not shown in the table for Domain 7 are Secure. The master can access resources that are controlled by MBC/MRC in other memory spaces when the master is in secure state.
2. Assign domain 7 for Cortex-A35.

How to determine COM port

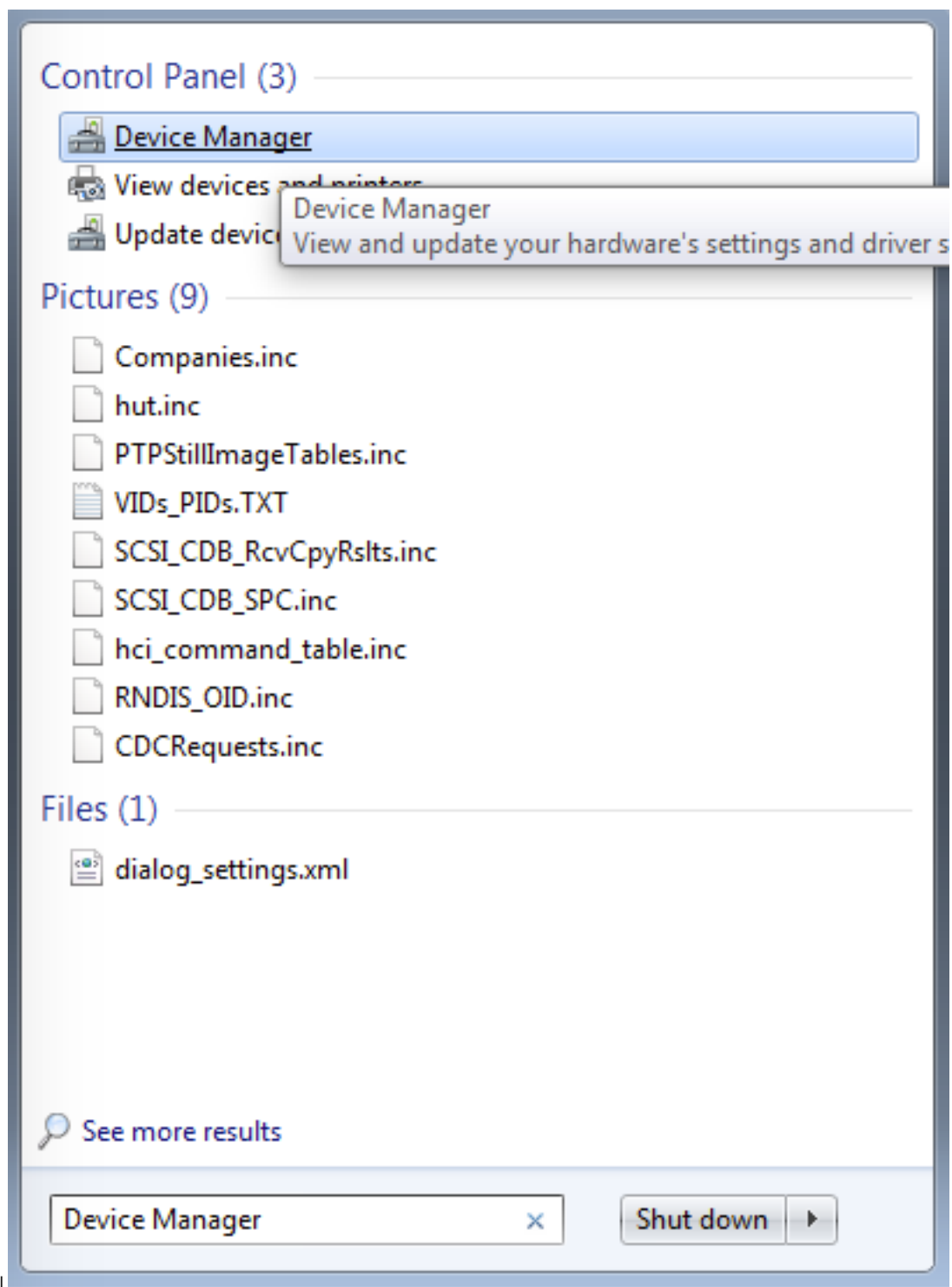
This section describes the steps to determine the debug COM port number of your NXP hardware development platform.

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

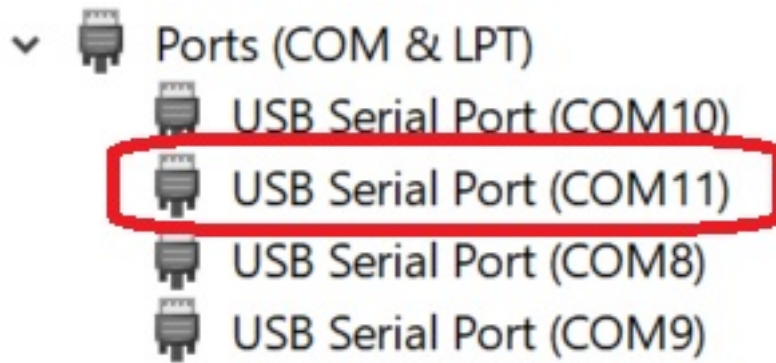
```
$ dmesg | grep "ttyUSB"
[503175.307873] usb 3-12: cp210x converter now attached to ttyUSB0
[503175.309372] usb 3-12: cp210x converter now attached to ttyUSB1
```

There are two ports, one is Cortex-A core debug console and the other is for Cortex M33.

2. **Windows:** To determine the COM port, open **Device Manager**. Click 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 are different for all the NXP boards.
 1. **USB-UART** interface



How to set up Windows/Linux host system

An MCUXpresso SDK build requires that some packages are installed on the host. Depending on the used host operating system, the following tools should be installed.

Linux:

- cmake

```
$ sudo apt-get install cmake $ # Check the version >= 3.0.x $ cmake --version
```

Windows:

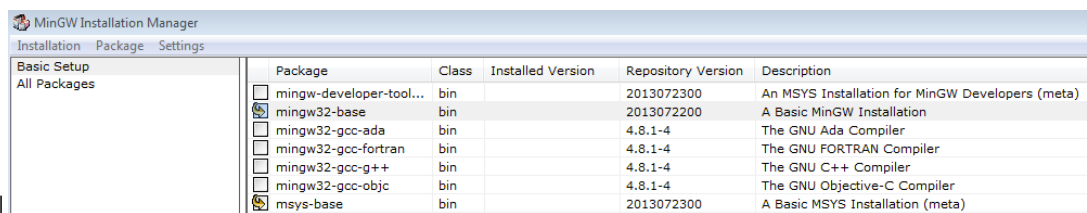
- MinGW

The Minimalist GNU for Windows OS (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 SDK does not utilize 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 sourceforge.net/projects/mingw/files/Installer/.
2. Run the installer. The recommended installation path is C:\MinGW, however, you may install to any location.

Note: The installation path should not contain any spaces.

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



4. Click ****Apply Changes**** in the ****Installation**** menu and follow the remaining instructions to complete the installation.

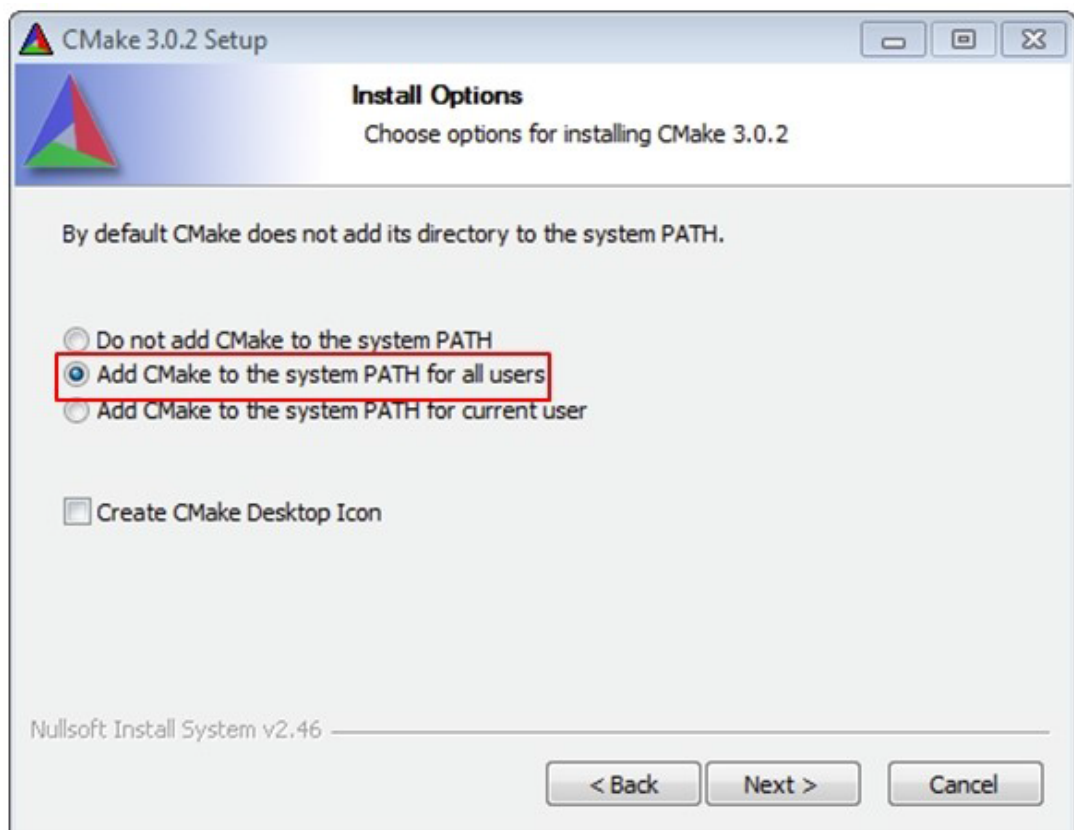
5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel** \> **System and Security** \> **System** \> **Advanced System Settings** in the **Environment Variables** section. The path is: `<mingw_install_dir>\bin`.`

Assuming the default installation path, `C:\MinGW``, an example is as shown in [Figure 3](how_to_set_up_windows_linux_host_system.md#ADDINGPATH). If the path is not set correctly, the toolchain does not work.

Note: If you have `C:\MinGW\msys\x.x\bin`` in your `PATH`` variable (as required by Kinetis SDK v2.10.0), remove it to ensure that the new GCC build system works correctly.

• CMake

1. Download CMake 3.0.x from www.cmake.org/cmake/resources/software.html.
2. While installing, ensure that the option **Add CMake to system PATH for all users** is selected. You can select install CMake 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. Reboot your system for the path changes to take effect.

1.3 Getting Started with MCUXpresso SDK GitHub

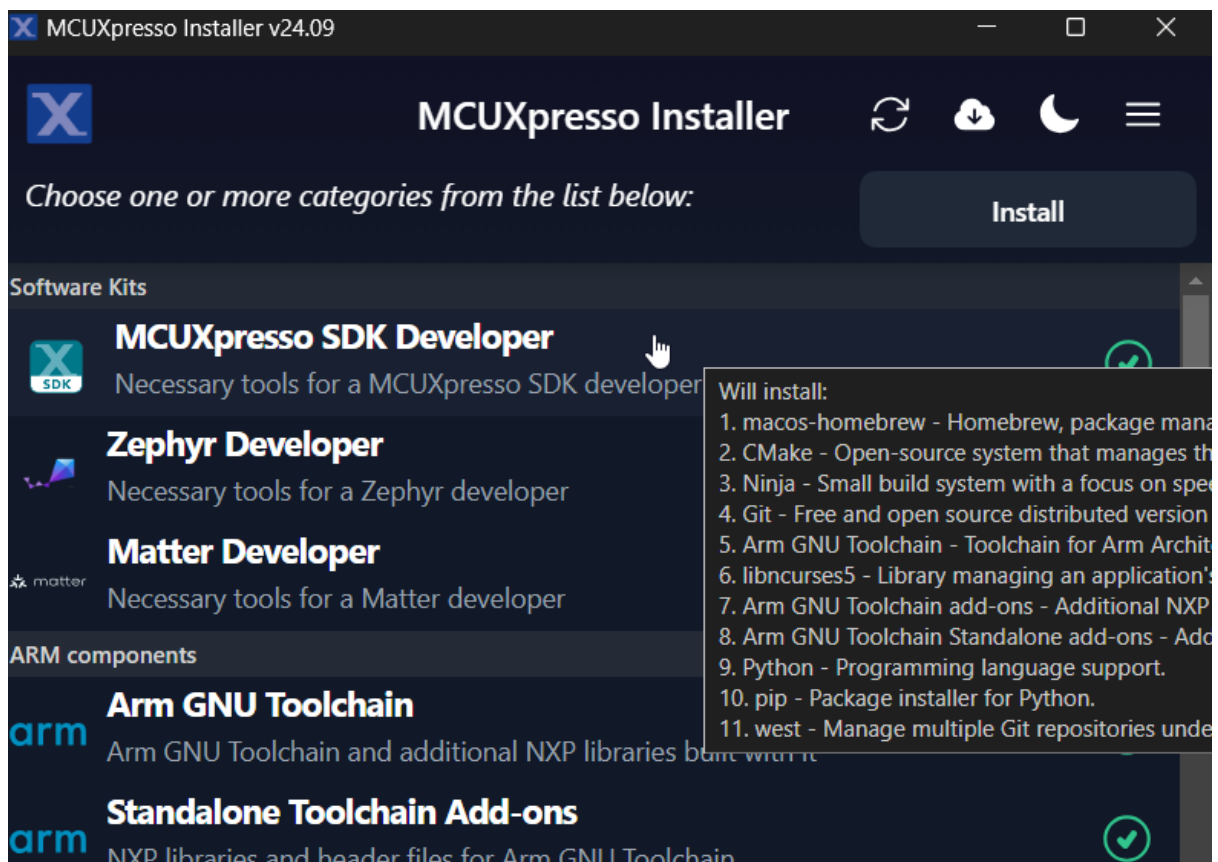
1.3.1 Getting Started with MCUXpresso SDK Repository

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.

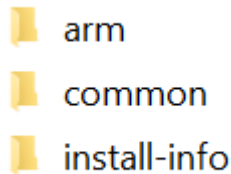
Toolchain MCUXpresso SDK supports all mainstream toolchains for embedded development. You can install your used or interested toolchains following the guides.

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_DIR	c:\NXP\zephyr-sdk-<version> for windows/opt/zephyr-sdk-<version> for Linux	– toolchain zephyr
CodeWarrior	CW_DIR	C:\Freescall\CW MCU v11.2 for windowsCodeWarrior is not supported with Linux	– toolchain code-warrior
Xtensa	XCC_DIR	C:\xtensa\XtDevTools\install\tools\RI-2023.11-win32\XtensaTools for windows/opt/xtensa/XtDevTools/install/tools/RI-2023.11-Linux/XtensaTools for Linux	– toolchain xtensa
NXP S32Compiler RISC-V Zen-V	RISCV-LVM_DIR	C:\riscv-llvm-win32_b298_b298_2024.08.12 for Windows/opt/riscv-llvm-Linux-x64_b298_b298_2024.08.12 for Linux	– toolchain riscv-llvm

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



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

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

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

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

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

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

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

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

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

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

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

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

Get MCUXpresso SDK Repo

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

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

# Update the west projects
cd mcuxpresso-sdk
west update

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

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

1. Navigate to the core directory:

```
cd mcuxsdk
```

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

```
python -m venv .venv

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

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

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

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

3. Install the required Python packages:

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

Explore Contents

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

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

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

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

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

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

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

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

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

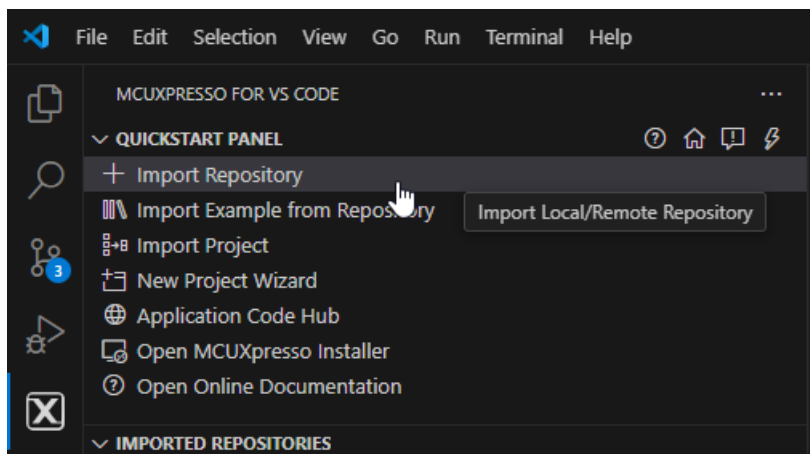
Run a demo using MCUXpresso for VS Code

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

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

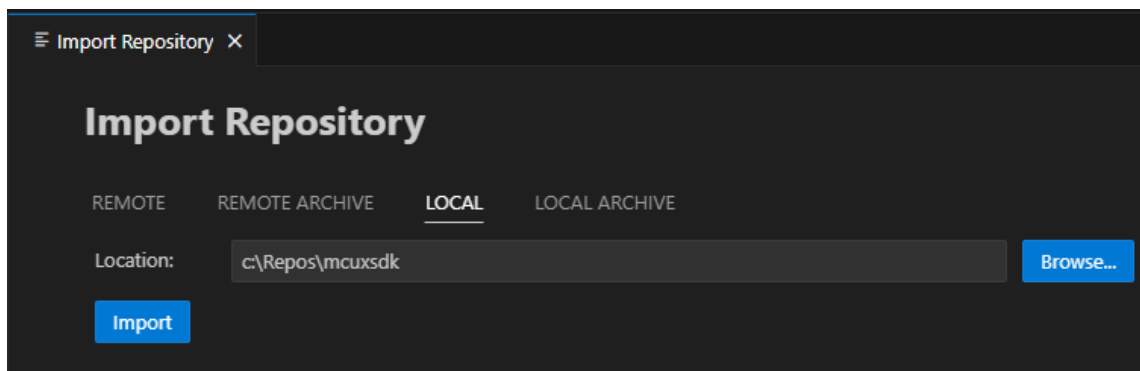
To build an example application:

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

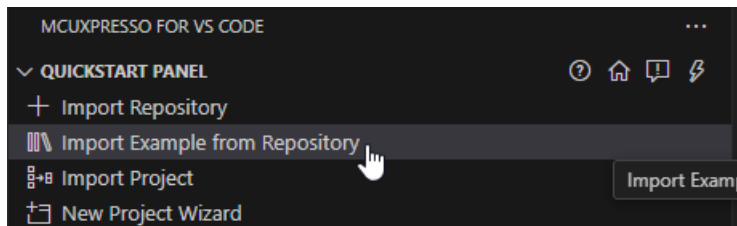


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

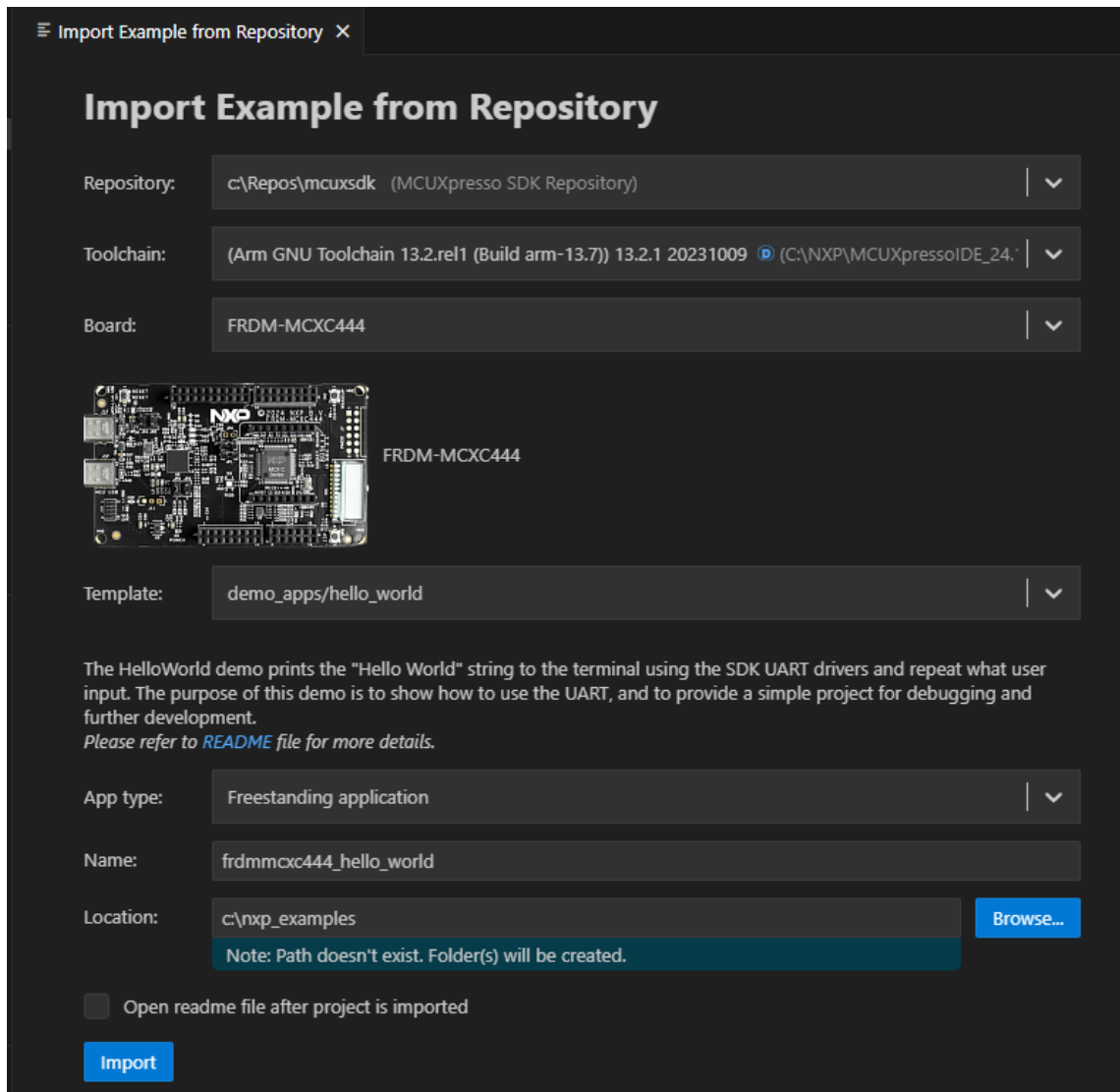
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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

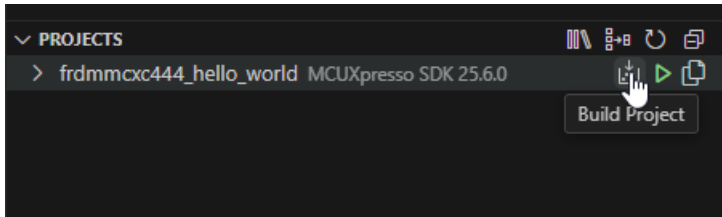


In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.

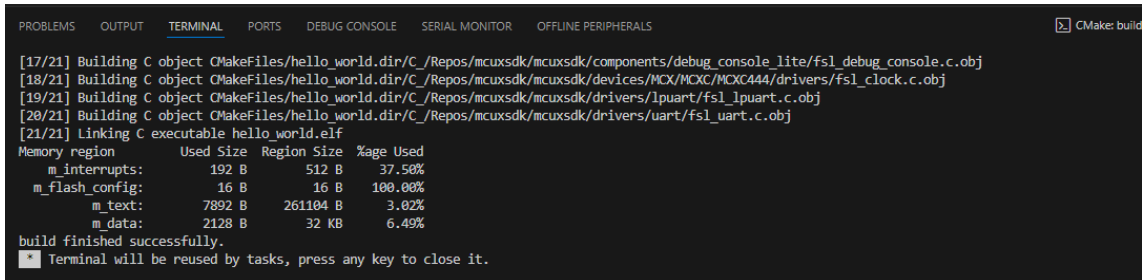


Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Freestanding applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Freestanding examples can be imported to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

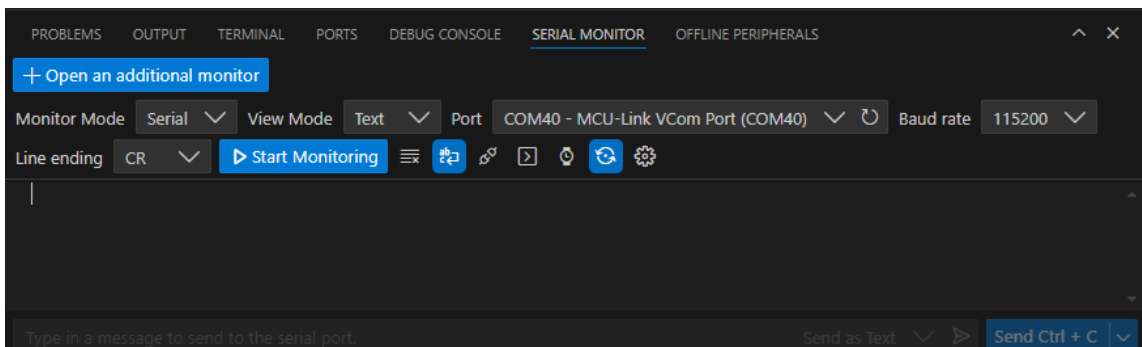


The integrated terminal will open at the bottom and will display the build output.

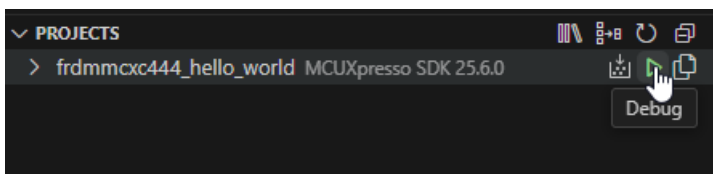


Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

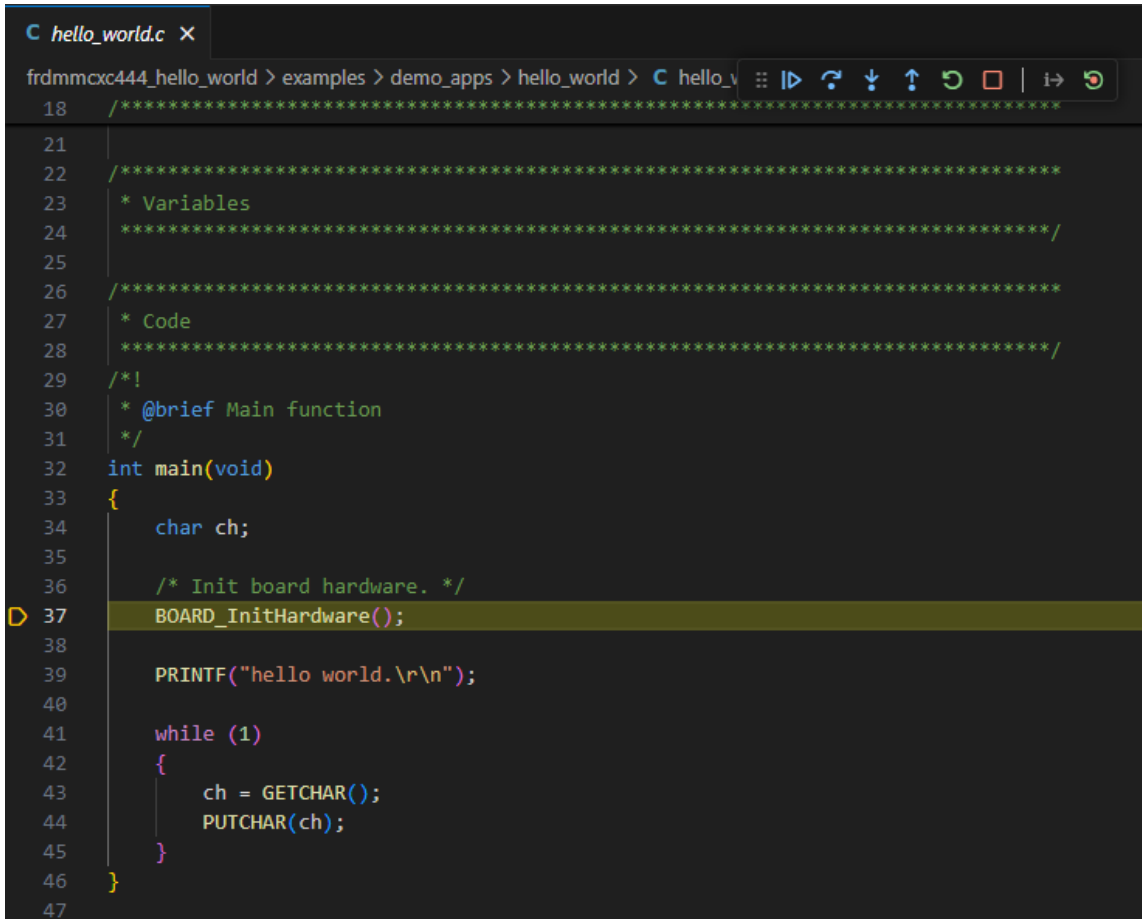
1. Open the **Serial Monitor** from the VS Code's integrated terminal. Select the VCom Port for your device and set the baud rate to 115200.



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.

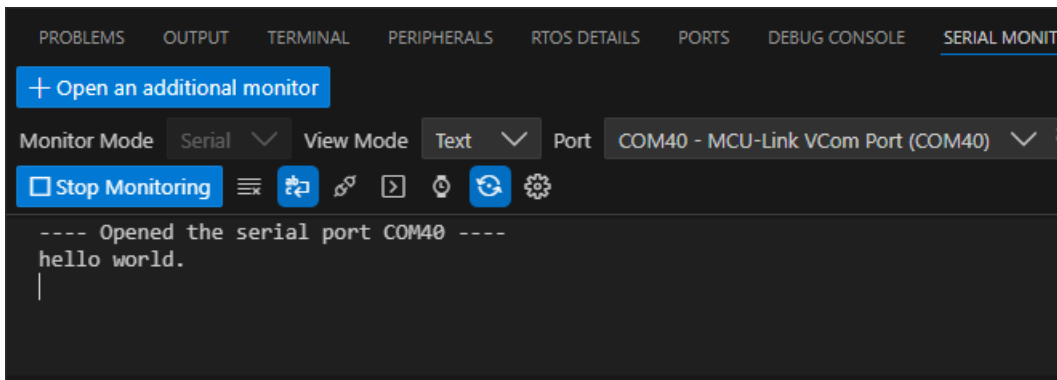


```

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)

```

↪ evkbbmimxrt1060]
INFO: [ 4][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbbmimxrt1170 -Dcore_id=cm4]
INFO: [ 5][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbbmimxrt1170 -Dcore_id=cm7]
INFO: [ 6][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkcmimxrt1060]
INFO: [ 7][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkmcimx7ulp]
...

```

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 evkbbmimxrt1170 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 evkbbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪ id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always

```

For more details, please refer to System build.

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

1. Run cmake configuration

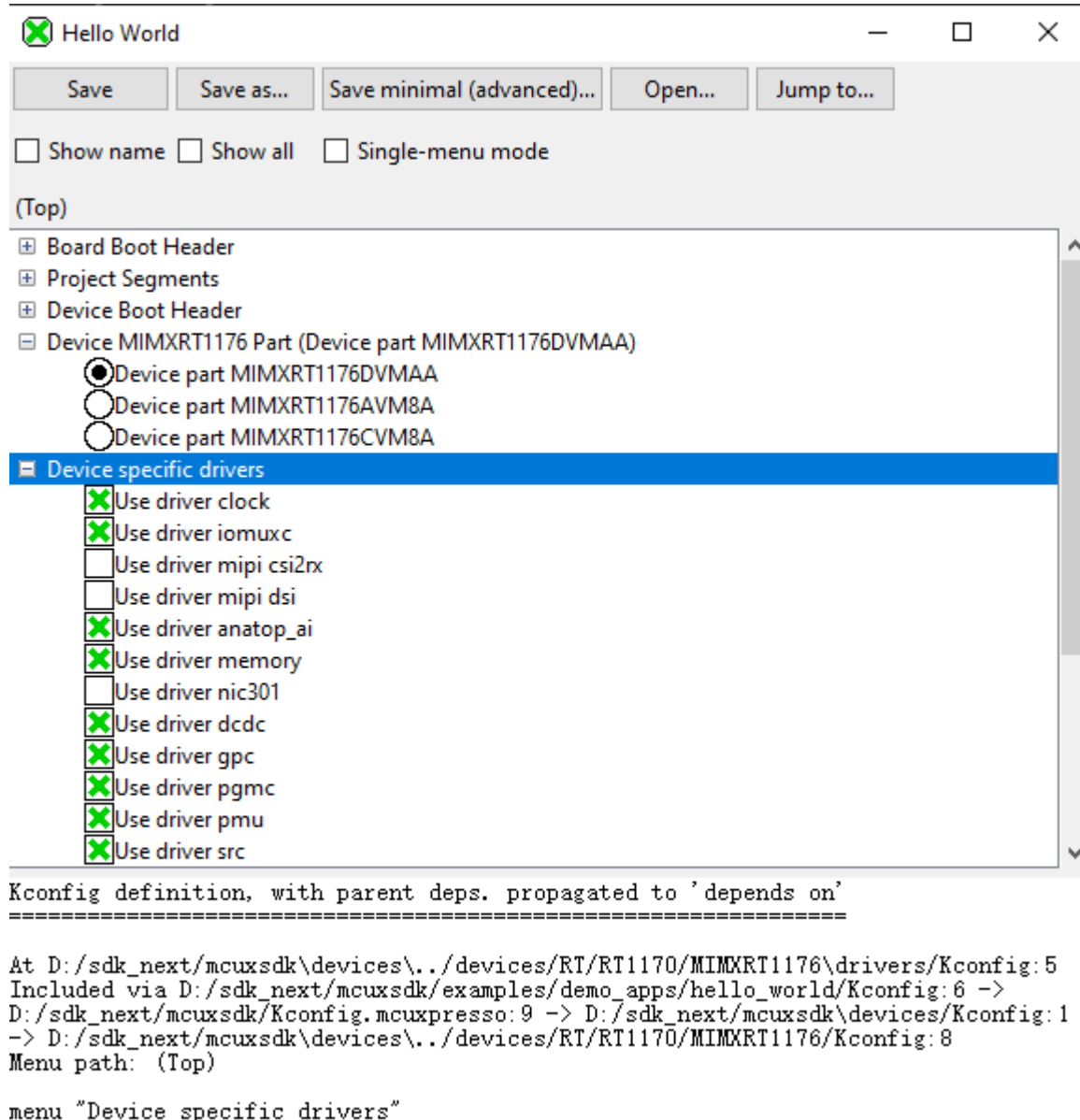
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

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

2. Run guiconfig target

```
west build -t guiconfig
```

Then you will get the Kconfig GUI launched, like



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

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

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

Flash the hello_world example:

```
west flash -r linkserver
```

Debug Start a gdb interface by following command:

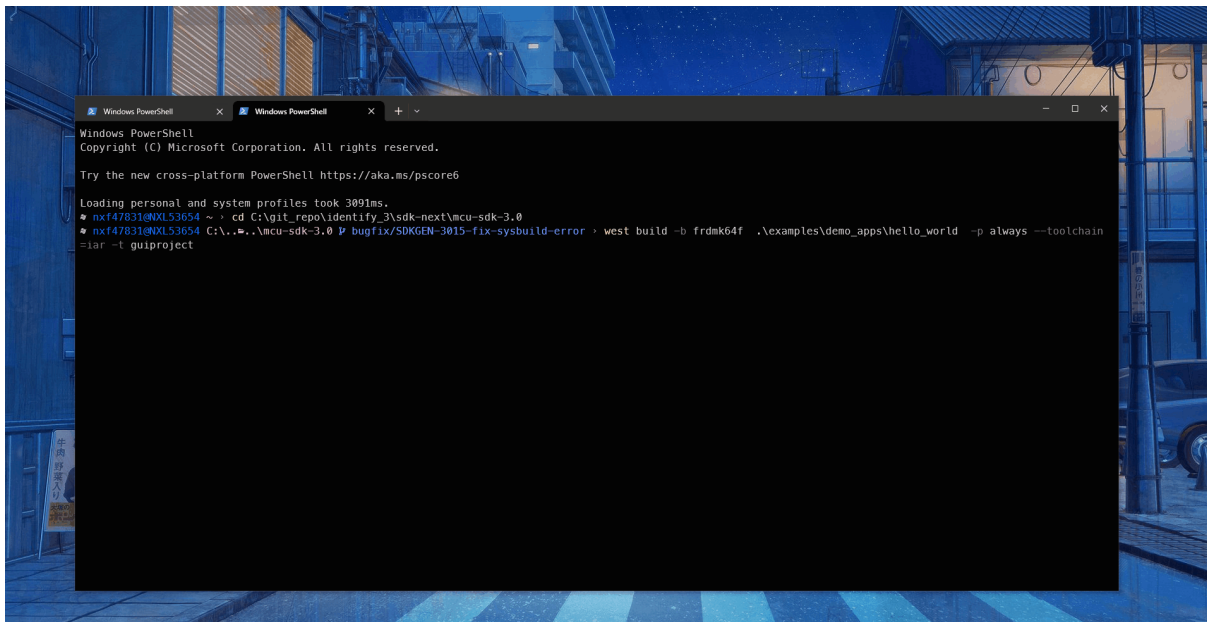
```
west debug -r linkserver
```

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

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

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

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



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

1.4 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.

- IAR Embedded Workbench for Arm, version is 9.60.4
- MCUXpresso for VS Code v25.06
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

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

De-velop-ment boards	MCU devices			
EVK-MIMX8	MIMX8UD3CVP08, MIMX8UD5DVK08, MIMX8US3DVP08, MIMX8UD7CVP08,	MIMX8UD3DVK08, MIMX8UD5DVP08, MIMX8US5CVP08, MIMX8UD7DVK08,	MIMX8UD3DVP08, MIMX8US3CVP08, MIMX8US5DVK08,	MIMX8UD5CVP08, MIMX8US3DVK08, MIMX8US5DVP08,
	MIMX8UD7DVP08			

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.

VoiceSpot VoiceSpot is a highly accurate, small memory and MIPS profile wake word engine supporting custom voice trigger words and phrases. MCUXpresso SDK version of VoiceSpot is trained to “Hey NXP” wake word only and has 25 hour trial timeout.

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”.

Multicore Multicore Software Development Kit

mbedtls mbedtls SSL/TLS library v2.x

llhttp HTTP parser llhttp

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

NatureDSP for FusionF1 Digital Signal Processing for Xtensa FusionF1 DSP Audio Engines.

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.

Fusion DSP may load wrong data from shared SRAM on Silicon A0.1 with specific command sequence

There is issue in the shared SRAM controller prefetch logic for silicon A0.1.

A back-2-back access to the same memory location without any “nop” cycle in between corrupts the prefetch buffer under the following conditions.

First cycle access is a read to address N, immediately followed by the second cycle-partial write (1 byte or halfword (16-bits)) to address N.

The partial write is not updated to prefetch buffer. Subsequent reading from the Shared SRAM address N returns incorrect data.

Real-Time Domain cannot normally resume from the power-down mode due to EdgeLock secure enclave (S400) failure

There is an issue in EdgeLock secure enclave (S400) during state restoring phase for silicon A0.1. When Real Time Domain (RTD) is about to enter the power-down mode, S400 is promoted to save the current state to the memory. However, once RTD wakes up, S400 encounters an error when trying to restore the state.

This error can cause S400 decide to reset RTD, which means RTD cannot normally resume from the power-down mode.

Flexcan_ping_pong_buffer_transfer case loses first 8 bytes data for armgcc flash_debug

To prevent flexcan_ping_pong_buffer_transfer case from losing the first 8 bytes data for armgcc flash_debug, apply the fix below.

```
a/flexcan/example/ping_pong_buffer_transfer/flexcan_ping_pong_buffer_transfer.c
+++ b/flexcan/example/ping_pong_buffer_transfer/flexcan_ping_pong_buffer_transfer.c
@@ -539,10 +539,11 @@ int main(void)
     else
     {
         /* Wait until Rx queue 1 full. */
-        while (rxQueueNum != 1U)
+        while (rxQueueNum == 0U)
         {
         };
-        rxQueueNum = 0;
+        if (rxQueueNum == 1)
+            rxQueueNum = 0;
         LOG_INFO("Read Rx MB from Queue 1.\r\n");
         for (i = 0; i < RX_QUEUE_BUFFER_SIZE; i++)
         {
```

Lpspi_interrupt_b2b_master/slave example transfer fail on iar/armgcc flash target

The lpspi_interrupt_b2b_master/slave example transfer fails to send the data from the master to the slave on iar/armgcc flash target.

DSP examples build failure in IAR 9.50.1

In IAR version 9.50.1, there are build issues when building dsp_examples. To build the dsp_examples, use an older version of IAR. This issue will be fixed in the next version of IAR tool.

DSP examples cannot boot the Fusion core

The examples in the dsp_examples folder can not run properly since they can not boot the Fusion DSP core. It can be fixed by removing line 633 from board.c: TRDC_MbcSetMemoryBlockConfig(TRDC, &mbcBlockConfig); under comment /* non secure state can access CGC0 (Pbridge0, slot 47) for HIFI4 DSP */. This will be fixed in the next release.

For the dsp_voice_spot_demo there is a wrong incbin.S file used. Please use the same file as for the rest of the DSP examples.

Examples `hello_world_ns`, `secure_faults_ns`, and `secure_faults_trdc_ns` have incorrect library path in GUI projects

When the affected examples are generated as GUI projects, the library linking the secure and non-secure worlds has an incorrect path set. This causes linking errors during project compilation.

Examples: `hello_world_ns`, `hello_world_s`, `secure_faults_ns`, `secure_faults_s`, `secure_faults_trdc_ns`, `secure_faults_trdc_s`

Affected toolchains: mdk, iar

Workaround: In the IDE project settings for the non-secure (`_ns`) project, find the linked library (named `hello_world_s_CMSE_lib.o`, or similar, depending on the example project) and replace the path to the library with `<build_directory>/<secure_world_project_folder>/<IDE>/`, replacing the subdirectory names with the build directory, the secure world project name, and IDE name.

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

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.

[MIMX8UD7](#)

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 Multicore

[*Multicore SDK*](#)

1.7.2 FreeMASTER

[*freemaster*](#)

1.7.3 FreeRTOS

[*FreeRTOS*](#)

Chapter 2

MIMX8UD7

2.1 ACMP: Analog Comparator Driver

`void ACMP_Init(CMP_Type *base, const acmp_config_t *config)`

Initializes the ACMP.

The default configuration can be got by calling `ACMP_GetDefaultConfig()`.

Parameters

- `base` – ACMP peripheral base address.
- `config` – Pointer to ACMP configuration structure.

`void ACMP_Deinit(CMP_Type *base)`

Deinitializes the ACMP.

Parameters

- `base` – ACMP peripheral base address.

`void ACMP_GetDefaultConfig(acmp_config_t *config)`

Gets the default configuration for ACMP.

This function initializes the user configuration structure to default value. The default value are:

Example:

```
config->enableHighSpeed = false;
config->enableInvertOutput = false;
config->useUnfilteredOutput = false;
config->enablePinOut = false;
config->enableHysteresisBothDirections = false;
config->hysteresisMode = kACMP_hysteresisMode0;
```

Parameters

- `config` – Pointer to ACMP configuration structure.

`void ACMP_Enable(CMP_Type *base, bool enable)`

Enables or disables the ACMP.

Parameters

- `base` – ACMP peripheral base address.
- `enable` – True to enable the ACMP.

`void ACMP_EnableLinkToDAC(CMP_Type *base, bool enable)`

Enables the link from CMP to DAC enable.

When this bit is set, the DAC enable/disable is controlled by the bit `CMP_C0[EN]` instead of `CMP_C1[DACEN]`.

Parameters

- `base` – ACMP peripheral base address.
- `enable` – Enable the feature or not.

`void ACMP_SetChannelConfig(CMP_Type *base, const acmp_channel_config_t *config)`

Sets the channel configuration.

Note that the plus/minus mux's setting is only valid when the positive/negative port's input isn't from DAC but from channel mux.

Example:

```
acmp_channel_config_t configStruct = {0};
configStruct.positivePortInput = kACMP_PortInputFromDAC;
configStruct.negativePortInput = kACMP_PortInputFromMux;
configStruct.minusMuxInput = 1U;
ACMP_SetChannelConfig(CMP0, &configStruct);
```

Parameters

- `base` – ACMP peripheral base address.
- `config` – Pointer to channel configuration structure.

`void ACMP_EnableDMA(CMP_Type *base, bool enable)`

Enables or disables DMA.

Parameters

- `base` – ACMP peripheral base address.
- `enable` – True to enable DMA.

`void ACMP_EnableWindowMode(CMP_Type *base, bool enable)`

Enables or disables window mode.

Parameters

- `base` – ACMP peripheral base address.
- `enable` – True to enable window mode.

`void ACMP_SetFilterConfig(CMP_Type *base, const acmp_filter_config_t *config)`

Configures the filter.

The filter can be enabled when the filter count is bigger than 1, the filter period is greater than 0 and the sample clock is from divided bus clock or the filter is bigger than 1 and the sample clock is from external clock. Detailed usage can be got from the reference manual.

Example:

```
acmp_filter_config_t configStruct = {0};
configStruct.filterCount = 5U;
configStruct.filterPeriod = 200U;
configStruct.enableSample = false;
ACMP_SetFilterConfig(CMP0, &configStruct);
```

Parameters

- `base` – ACMP peripheral base address.

- `config` – Pointer to filter configuration structure.

`void ACMP_SetDACConfig(CMP_Type *base, const acmp_dac_config_t *config)`

Configures the internal DAC.

Example:

```
acmp_dac_config_t configStruct = {0};
configStruct.referenceVoltageSource = kACMP_VrefSourceVin1;
configStruct.DACValue = 20U;
configStruct.enableOutput = false;
configStruct.workMode = kACMP_DACWorkLowSpeedMode;
ACMP_SetDACConfig(CMP0, &configStruct);
```

Parameters

- `base` – ACMP peripheral base address.
- `config` – Pointer to DAC configuration structure. “NULL” is for disabling the feature.

`void ACMP_SetRoundRobinConfig(CMP_Type *base, const acmp_round_robin_config_t *config)`

Configures the round robin mode.

Example:

```
acmp_round_robin_config_t configStruct = {0};
configStruct.fixedPort = kACMP_FixedPlusPort;
configStruct.fixedChannelNumber = 3U;
configStruct.checkerChannelMask = 0xF7U;
configStruct.sampleClockCount = 0U;
configStruct.delayModulus = 0U;
ACMP_SetRoundRobinConfig(CMP0, &configStruct);
```

Parameters

- `base` – ACMP peripheral base address.
- `config` – Pointer to round robin mode configuration structure. “NULL” is for disabling the feature.

`void ACMP_SetRoundRobinPreState(CMP_Type *base, uint32_t mask)`

Defines the pre-set state of channels in round robin mode.

Note: The pre-state has different circuit with get-round-robin-result in the SOC even though they are same bits. So get-round-robin-result can't return the same value as the value are set by pre-state.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – Mask of round robin channel index. Available range is channel0:0x01 to channel7:0x80.

`static inline uint32_t ACMP_GetRoundRobinStatusFlags(CMP_Type *base)`

Gets the channel input changed flags in round robin mode.

Parameters

- `base` – ACMP peripheral base address.

Returns

Mask of channel input changed asserted flags. Available range is channel0:0x01 to channel7:0x80.

`void ACMP_ClearRoundRobinStatusFlags(CMP_Type *base, uint32_t mask)`

Clears the channel input changed flags in round robin mode.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – Mask of channel index. Available range is channel0:0x01 to channel7:0x80.

`static inline uint32_t ACMP_GetRoundRobinResult(CMP_Type *base)`

Gets the round robin result.

Note that the set-pre-state has different circuit with get-round-robin-result in the SOC even though they are same bits. So `ACMP_GetRoundRobinResult()` can't return the same value as the value are set by `ACMP_SetRoundRobinPreState`.

Parameters

- `base` – ACMP peripheral base address.

Returns

Mask of round robin channel result. Available range is channel0:0x01 to channel7:0x80.

`void ACMP_EnableInterrupts(CMP_Type *base, uint32_t mask)`

Enables interrupts.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – Interrupts mask. See “`_acmp_interrupt_enable`”.

`void ACMP_DisableInterrupts(CMP_Type *base, uint32_t mask)`

Disables interrupts.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – Interrupts mask. See “`_acmp_interrupt_enable`”.

`uint32_t ACMP_GetStatusFlags(CMP_Type *base)`

Gets status flags.

Parameters

- `base` – ACMP peripheral base address.

Returns

Status flags asserted mask. See “`_acmp_status_flags`”.

`void ACMP_ClearStatusFlags(CMP_Type *base, uint32_t mask)`

Clears status flags.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – Status flags mask. See “`_acmp_status_flags`”.

`void ACMP_SetDiscreteModeConfig(CMP_Type *base, const acmp_discrete_mode_config_t *config)`

Configure the discrete mode.

Configure the discrete mode when supporting 3V domain with 1.8V core.

Parameters

- base – ACMP peripheral base address.
- config – Pointer to configuration structure. See “acmp_discrete_mode_config_t”.

void ACMP_GetDefaultDiscreteModeConfig(*acmp_discrete_mode_config_t* *config)

Get the default configuration for discrete mode setting.

Parameters

- config – Pointer to configuration structure to be restored with the setting values.

FSL_ACMP_DRIVER_VERSION

ACMP driver version 2.3.0.

enum _acmp_interrupt_enable

Interrupt enable/disable mask.

Values:

enumerator kACMP_OutputRisingInterruptEnable

Enable the interrupt when comparator outputs rising.

enumerator kACMP_OutputFallingInterruptEnable

Enable the interrupt when comparator outputs falling.

enumerator kACMP_RoundRobinInterruptEnable

Enable the Round-Robin interrupt.

enum _acmp_status_flags

Status flag mask.

Values:

enumerator kACMP_OutputRisingEventFlag

Rising-edge on compare output has occurred.

enumerator kACMP_OutputFallingEventFlag

Falling-edge on compare output has occurred.

enumerator kACMP_OutputAssertEventFlag

Return the current value of the analog comparator output.

enum _acmp_offset_mode

Comparator hard block offset control.

If OFFSET level is 1, then there is no hysteresis in the case of positive port input crossing negative port input in the positive direction (or negative port input crossing positive port input in the negative direction). Hysteresis still exists for positive port input crossing negative port input in the falling direction. If OFFSET level is 0, then the hysteresis selected by *acmp_hysteresis_mode_t* is valid for both directions.

Values:

enumerator kACMP_OffsetLevel0

The comparator hard block output has level 0 offset internally.

enumerator kACMP_OffsetLevel1

The comparator hard block output has level 1 offset internally.

enum _acmp_hysteresis_mode

Comparator hard block hysteresis control.

See chip data sheet to get the actual hysteresis value with each level.

Values:

enumerator kACMP_HysteresisLevel0

Offset is level 0 and Hysteresis is level 0.

enumerator kACMP_HysteresisLevel1

Offset is level 0 and Hysteresis is level 1.

enumerator kACMP_HysteresisLevel2

Offset is level 0 and Hysteresis is level 2.

enumerator kACMP_HysteresisLevel3

Offset is level 0 and Hysteresis is level 3.

enum _acmp_reference_voltage_source

CMP Voltage Reference source.

Values:

enumerator kACMP_VrefSourceVin1

Vin1 is selected as resistor ladder network supply reference Vin.

enumerator kACMP_VrefSourceVin2

Vin2 is selected as resistor ladder network supply reference Vin.

enum _acmp_port_input

Port input source.

Values:

enumerator kACMP_PortInputFromDAC

Port input from the 8-bit DAC output.

enumerator kACMP_PortInputFromMux

Port input from the analog 8-1 mux.

enum _acmp_fixed_port

Fixed mux port.

Values:

enumerator kACMP_FixedPlusPort

Only the inputs to the Minus port are swept in each round.

enumerator kACMP_FixedMinusPort

Only the inputs to the Plus port are swept in each round.

enum _acmp_dac_work_mode

Internal DAC's work mode.

Values:

enumerator kACMP_DACWorkLowSpeedMode

DAC is selected to work in low speed and low power mode.

enumerator kACMP_DACWorkHighSpeedMode

DAC is selected to work in high speed high power mode.

typedef enum _acmp_offset_mode acmp_offset_mode_t

Comparator hard block offset control.

If OFFSET level is 1, then there is no hysteresis in the case of positive port input crossing negative port input in the positive direction (or negative port input crossing positive port input in the negative direction). Hysteresis still exists for positive port input crossing negative port input in the falling direction. If OFFSET level is 0, then the hysteresis selected by acmp_hysteresis_mode_t is valid for both directions.

typedef enum *_acmp_hysteresis_mode* acmp_hysteresis_mode_t

Comparator hard block hysteresis control.

See chip data sheet to get the actual hysteresis value with each level.

typedef enum *_acmp_reference_voltage_source* acmp_reference_voltage_source_t

CMP Voltage Reference source.

typedef enum *_acmp_port_input* acmp_port_input_t

Port input source.

typedef enum *_acmp_fixed_port* acmp_fixed_port_t

Fixed mux port.

typedef enum *_acmp_dac_work_mode* acmp_dac_work_mode_t

Internal DAC's work mode.

typedef struct *_acmp_config* acmp_config_t

Configuration for ACMP.

typedef struct *_acmp_channel_config* acmp_channel_config_t

Configuration for channel.

The comparator's port can be input from channel mux or DAC. If port input is from channel mux, detailed channel number for the mux should be configured.

typedef struct *_acmp_filter_config* acmp_filter_config_t

Configuration for filter.

typedef struct *_acmp_dac_config* acmp_dac_config_t

Configuration for DAC.

typedef struct *_acmp_round_robin_config* acmp_round_robin_config_t

Configuration for round robin mode.

typedef struct *_acmp_discrete_mode_config* acmp_discrete_mode_config_t

Configuration for discrete mode.

CMP_C0_CFX_MASK

The mask of status flags cleared by writing 1.

CMP_C1_CHNn_MASK

CMP_C2_CHnF_MASK

struct *_acmp_config*

#include <fsl_acmp.h> Configuration for ACMP.

Public Members

acmp_offset_mode_t offsetMode

Offset mode.

acmp_hysteresis_mode_t hysteresisMode

Hysteresis mode.

bool enableHighSpeed

Enable High Speed (HS) comparison mode.

bool enableInvertOutput

Enable inverted comparator output.

bool useUnfilteredOutput

Set compare output(COUT) to equal COUTA(true) or COUT(false).

bool enablePinOut

The comparator output is available on the associated pin.

struct __acmp_channel_config

#include <fsl_acmp.h> Configuration for channel.

The comparator's port can be input from channel mux or DAC. If port input is from channel mux, detailed channel number for the mux should be configured.

Public Members

acmp_port_input_t positivePortInput

Input source of the comparator's positive port.

uint32_t plusMuxInput

Plus mux input channel(0~7).

acmp_port_input_t negativePortInput

Input source of the comparator's negative port.

uint32_t minusMuxInput

Minus mux input channel(0~7).

struct __acmp_filter_config

#include <fsl_acmp.h> Configuration for filter.

Public Members

bool enableSample

Using external SAMPLE as sampling clock input, or using divided bus clock.

uint32_t filterCount

Filter Sample Count. Available range is 1-7, 0 would cause the filter disabled.

uint32_t filterPeriod

Filter Sample Period. The divider to bus clock. Available range is 0-255.

struct __acmp_dac_config

#include <fsl_acmp.h> Configuration for DAC.

Public Members

acmp_reference_voltage_source_t referenceVoltageSource

Supply voltage reference source.

uint32_t DACValue

Value for DAC Output Voltage. Available range is 0-255.

bool enableOutput

Enable the DAC output.

struct __acmp_round_robin_config

#include <fsl_acmp.h> Configuration for round robin mode.

Public Members

acmp_fixed_port_t fixedPort

Fixed mux port.

uint32_t fixedChannelNumber

Indicates which channel is fixed in the fixed mux port.

uint32_t checkerChannelMask

Mask of checker channel index. Available range is channel0:0x01 to channel7:0x80 for round-robin checker.

uint32_t sampleClockCount

Specifies how many round-robin clock cycles(0~3) later the sample takes place.

uint32_t delayModulus

Comparator and DAC initialization delay modulus.

struct _acmp_discrete_mode_config

#include <fsl_acmp.h> Configuration for discrete mode.

Public Members

bool enablePositiveChannelDiscreteMode

Positive Channel Continuous Mode Enable. By default, the continuous mode is used.

bool enableNegativeChannelDiscreteMode

Negative Channel Continuous Mode Enable. By default, the continuous mode is used.

2.2 Battery-Backed Non-Secure Module

void BBNSM_Init(BBNSM_Type *base)

Init the BBNSM section.

Parameters

- base – BBNSM peripheral base address

void BBNSM_Deinit(BBNSM_Type *base)

Deinit the BBNSM section.

Parameters

- base – BBNSM peripheral base address

FSL_BBNSM_DRIVER_VERSION

Version 2.0.0

enum _bbnsm_interrupts

List of BBNSM interrupts.

Values:

enumerator kBBNSM_RTC_AlarmInterrupt

RTC time alarm interrupt

enumerator kBBNSM_RTC_RolloverInterrupt

RTC rollover interrupt

enum *_bbnsm_status_flags*

List of BBNSM flags.

Values:

enumerator *kBBNSM_RTC_AlarmInterruptFlag*
RTC time alarm interrupt flag

enumerator *kBBNSM_RTC_RolloverInterruptFlag*
RTC rollover interrupt flag

enumerator *kBBNSM_PWR_ON_InterruptFlag*
power on interrupt flag

enumerator *kBBNSM_PWR_OFF_InterruptFlag*
power off interrupt flag

enumerator *kBBNSM_EMG_OFF_InterruptFlag*
emergency power off interrupt flag

typedef enum *_bbnsm_interrupts* *bbnsm_interrupts_t*
List of BBNSM interrupts.

typedef enum *_bbnsm_status_flags* *bbnsm_status_flags_t*
List of BBNSM flags.

typedef struct *_bbnsm_rtc_config* *bbnsm_rtc_config_t*
BBNSM config structure.

This structure holds the configuration settings for the BBNSM peripheral. To initialize this structure to reasonable defaults, call the *BBNSM_RTC_GetDefaultConfig()* function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

void *BBNSM_RTC_Init*(*BBNSM_Type* *base, const *bbnsm_rtc_config_t* *config)
Un-gates the BBNSM clock and configures the peripheral for basic operation.

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

Parameters

- base – BBNSM peripheral base address
- config – Pointer to the user's BBNSM rtc configuration structure.

void *BBNSM_RTC_Deinit*(*BBNSM_Type* *base)
Stops the RTC timer.

Parameters

- base – BBNSM peripheral base address

void *BBNSM_RTC_GetDefaultConfig*(*bbnsm_rtc_config_t* *config)
Fills in the BBNSM RTC config struct with the default settings.

The default values are as follows.

```
config->rtccalenable = false;  
config->rtccalvalue = 0U;
```

Parameters

- config – Pointer to the user's BBNSM configuration structure.

`status_t BBNSM_RTC_SetAlarm(BBNSM_Type *base, uint32_t alarmSeconds)`

Sets the BBNSM 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. Please note, that RTC alarm has limited resolution because only 32 most significant bits of RTC counter are compared to RTC Alarm register. If the alarm time is beyond RTC resolution, the function does not set the alarm and returns an error.

Parameters

- `base` – BBNSM peripheral base address
- `alarmSeconds` –

Returns

`kStatus_Success`: success in setting the BBNSM RTC 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

`uint32_t BBNSM_RTC_GetAlarm(BBNSM_Type *base)`

Returns the BBNSM RTC alarm time.

Parameters

- `base` – BBNSM peripheral base address

`struct _bbnsn_rtc_config`

`#include <fsl_bbnsn.h>` BBNSM config structure.

This structure holds the configuration settings for the BBNSM peripheral. To initialize this structure to reasonable defaults, call the `BBNSM_RTC_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

`bool rtcCalEnable`

true: RTC calibration mechanism is enabled; false: No calibration is used

`uint32_t rtcCalValue`

Defines signed calibration value for RTC; This is a 5-bit 2's complement value, range from -16 to +15

2.3 CACHE: CACHE Memory Controller

`uint32_t CACHE64_GetInstanceByAddr(uint32_t address)`

brief Returns an instance number given physical memory address.

param address The physical memory address.

Returns

CACHE64_CTRL instance number starting from 0.

`void CACHE64_EnableCache(CACHE64_CTRL_Type *base)`

Enables the cache.

Parameters

- `base` – CACHE64_CTRL peripheral base address.

void CACHE64_DisableCache(CACHE64_CTRL_Type *base)

Disables the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_InvalidateCache(CACHE64_CTRL_Type *base)

Invalidates the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_InvalidateCacheByRange(uint32_t address, uint32_t size_byte)

Invalidates cache by range.

Note: Address and size should be aligned to “CACHE64_LINESIZE_BYTE”. The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE 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 of cache.
- size_byte – size of the memory to be invalidated, should be larger than 0.

void CACHE64_CleanCache(CACHE64_CTRL_Type *base)

Cleans the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_CleanCacheByRange(uint32_t address, uint32_t size_byte)

Cleans cache by range.

Note: Address and size should be aligned to “CACHE64_LINESIZE_BYTE”. The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE 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 of cache.
- size_byte – size of the memory to be cleaned, should be larger than 0.

void CACHE64_CleanInvalidateCache(CACHE64_CTRL_Type *base)

Cleans and invalidates the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_CleanInvalidateCacheByRange(uint32_t address, uint32_t size_byte)

Cleans and invalidate cache by range.

Note: Address and size should be aligned to “CACHE64_LINESIZE_BYTE”. The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE 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 of cache.
- size_byte – size of the memory to be Cleaned and Invalidated, should be larger than 0.

static inline void ICACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)

Invalidates instruction cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. 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, should be larger than 0.

static inline void DCACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)

Invalidates data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. 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, should be larger than 0.

static inline void DCACHE_CleanByRange(uint32_t address, uint32_t size_byte)

Clean data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. 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, should be larger than 0.

static inline void DCACHE_CleanInvalidateByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. 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, should be larger than 0.

FSL_CACHE_DRIVER_VERSION
cache driver version.

CACHE64_LINESIZE_BYTE
cache line size.

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

2.5 casper_driver

FSL_CASPER_DRIVER_VERSION
CASPER driver version. Version 2.2.4.

Current version: 2.2.4

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Bug fix KPSDK-24531 double_scalar_multiplication() result may be all zeroes for some specific input
- Version 2.0.2
 - Bug fix KPSDK-25015 CASPER_MEMCPY hard-fault on LPC55xx when both source and destination buffers are outside of CASPER_RAM
- Version 2.0.3
 - Bug fix KPSDK-28107 RSUB, FILL and ZERO operations not implemented in enum _casper_operation.
- Version 2.0.4
 - For GCC compiler, enforce O1 optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires -fno-strict-aliasing.
- Version 2.0.5
 - Fix sign-compare warning.

- Version 2.0.6
 - Fix IAR Pa082 warning.
- Version 2.0.7
 - Fix MISRA-C 2012 issue.
- Version 2.0.8
 - Add feature macro for CASPER_RAM_OFFSET.
- Version 2.0.9
 - Remove unused function Jac_oncurve().
 - Fix ECC384 build.
- Version 2.0.10
 - Fix MISRA-C 2012 issue.
- Version 2.1.0
 - Add ECC NIST P-521 elliptic curve.
- Version 2.2.0
 - Rework driver to support multiple curves at once.
- Version 2.2.1
 - Fix MISRA-C 2012 issue.
- Version 2.2.2
 - Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE
- Version 2.2.3
 - Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.
- Version 2.2.4
 - Fix MISRA-C 2012 issue.

enum _casper_operation

CASPER operation.

Values:

enumerator kCASPER_OpMul6464NoSum

enumerator kCASPER_OpMul6464Sum

Walking 1 or more of J loop, doing $r=a*b$ using $64x64=128$

enumerator kCASPER_OpMul6464FullSum

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64x64=128$, but assume inner j loop

enumerator kCASPER_OpMul6464Reduce

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64x64=128$, but sum all of w.

enumerator kCASPER_OpAdd64

Walking 1 or more of J loop, doing $c,r[-1]=r+a*b$ using $64x64=128$, but skip 1st write

enumerator kCASPER_OpSub64

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

enumerator kCASPER_OpDouble64

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

enumerator kCASPER_OpXor64

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

enumerator kCASPER_OpRSub64

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

enumerator kCASPER_OpShiftLeft32

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

enumerator kCASPER_OpShiftRight32

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

enumerator kCASPER_OpCopy

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

enumerator kCASPER_OpRemask

Copy from ABoff to resoff, 64b at a time

enumerator kCASPER_OpFill

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

enumerator kCASPER_OpZero

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

enumerator kCASPER_OpCompare

Fill RESOFF using 64 bits at a time of 0s

enumerator kCASPER_OpCompareFast

Compare two arrays, running all the way to the end

enum _casper_algo_t

Algorithm used for CASPER operation.

Values:

enumerator kCASPER_ECC_P256

ECC_P256

enumerator kCASPER_ECC_P384

ECC_P384

enumerator kCASPER_ECC_P521

ECC_P521

Values:

enumerator kCASPER_RamOffset_Result

enumerator kCASPER_RamOffset_Base

enumerator kCASPER_RamOffset_TempBase

enumerator kCASPER_RamOffset_Modulus

enumerator kCASPER_RamOffset_M64

`typedef enum _casper_operation casper_operation_t`
CASPER operation.

`typedef enum _casper_algo_t casper_algo_t`
Algorithm used for CASPER operation.

`void CASPER_Init(CASPER_Type *base)`
Enables clock and disables reset for CASPER peripheral.
Enable clock and disable reset for CASPER.

Parameters

- base – CASPER base address

`void CASPER_Deinit(CASPER_Type *base)`
Disables clock for CASPER peripheral.
Disable clock and enable reset.

Parameters

- base – CASPER base address

`CASPER_CP`

`CASPER_CP_CTRL0`

`CASPER_CP_CTRL1`

`CASPER_CP_LOADER`

`CASPER_CP_STATUS`

`CASPER_CP_INTENSET`

`CASPER_CP_INTENCLR`

`CASPER_CP_INTSTAT`

`CASPER_CP_AREG`

`CASPER_CP_BREG`

`CASPER_CP_CREG`

`CASPER_CP_DREG`

`CASPER_CP_RES0`

`CASPER_CP_RES1`

`CASPER_CP_RES2`

`CASPER_CP_RES3`

`CASPER_CP_MASK`

`CASPER_CP_REMASK`

`CASPER_CP_LOCK`

`CASPER_CP_ID`

`CASPER_Wr32b(value, off)`

`CASPER_Wr64b(value, off)`

CASPER_Rd32b(off)

N_wordlen_max

2.6 casper_driver_pkha

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

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

This function performs modular exponentiation.

Parameters

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

void CASPER_ecc_init(casper_algo_t curve)

Initialize prime modulus mod in Casper memory .

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

Parameters

- curve – elliptic curve algorithm

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

Performs ECC secp256r1 point single scalar multiplication.

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

Parameters

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

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

Performs ECC secp256r1 point double scalar multiplication.

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

Parameters

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

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

Performs ECC secp384r1 point single scalar multiplication.

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

Parameters

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

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

Performs ECC secp384r1 point double scalar multiplication.

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

Parameters

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

- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.
- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

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

Performs ECC secp521r1 point single scalar multiplication.

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

Parameters

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

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

Performs ECC secp521r1 point double scalar multiplication.

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

Parameters

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

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

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

2.7 Clock

enum _clock_name

Clock name used to get clock frequency.

Values:

enumerator kCLOCK_Cm33CorePlatClk

RTD : CM33 Core/Platform clock

enumerator kCLOCK_Cm33BusClk

RTD : CM33 Bus clock

enumerator kCLOCK_Cm33SlowClk

RTD : CM33 Slow clock

enumerator kCLOCK_FusionDspCorePlatClk

RTD : FusionF1 DSP Core/Platform clock

enumerator kCLOCK_FusionDspBusClk

RTD : FusionF1 DSP Bus clock

enumerator kCLOCK_FusionDspSlowClk

RTD : FusionF1 DSP Slow clock

enumerator kCLOCK_XbarBusClk

AD : XBAR Bus clock

enumerator kCLOCK_HifiDspClk

LPAV: HIFI4 clock

enumerator kCLOCK_HifiNicPlatClk

LPAV: NIC HIFI clock

enumerator kCLOCK_NicLpavAxiClk

LPAV: NIC LPAV AXI clock

enumerator kCLOCK_NicLpavAhbClk

LPAV: NIC LPAV AHB clock

enumerator kCLOCK_NicLpavBusClk

LPAV: LPAV Bus clock

enumerator kCLOCK_DdrClk

LPAV: DDR clock

enumerator kCLOCK_SysOscClk

CGC system OSC clock. (SYSOSC)

enumerator kCLOCK_FroClk

CGC FRO 192MHz clock.

enumerator kCLOCK_LpOscClk

CGC LPOSC clock. (LPOSC)

enumerator kCLOCK_RtcOscClk

CGC RTC OSC clock. (RTCOSC)

enumerator kCLOCK_LvdsClk

LVDS pad input clock frequency.

enumerator kCLOCK_RtdFroDiv1Clk
FRODIV1_CLK in RTD.

enumerator kCLOCK_RtdFroDiv2Clk
FRODIV2_CLK in RTD.

enumerator kCLOCK_RtdFroDiv3Clk
FRODIV3_CLK in RTD.

enumerator kCLOCK_RtdSysOscDiv1Clk
SOSCDIV1_CLK in RTD.

enumerator kCLOCK_RtdSysOscDiv2Clk
SOSCDIV2_CLK in RTD.

enumerator kCLOCK_RtdSysOscDiv3Clk
SOSCDIV3_CLK in RTD.

enumerator kCLOCK_AdFroDiv1Clk
FRODIV1_CLK in AD.

enumerator kCLOCK_AdFroDiv2Clk
FRODIV2_CLK in AD.

enumerator kCLOCK_AdFroDiv3Clk
FRODIV3_CLK in AD.

enumerator kCLOCK_AdSysOscDiv1Clk
SOSCDIV1_CLK in AD.

enumerator kCLOCK_AdSysOscDiv2Clk
SOSCDIV2_CLK in AD.

enumerator kCLOCK_AdSysOscDiv3Clk
SOSCDIV3_CLK in AD.

enumerator kCLOCK_LpavFroDiv1Clk
FRODIV1_CLK in LPAV.

enumerator kCLOCK_LpavFroDiv2Clk
FRODIV2_CLK in LPAV.

enumerator kCLOCK_LpavFroDiv3Clk
FRODIV3_CLK in LPAV.

enumerator kCLOCK_LpavSysOscDiv1Clk
SOSCDIV1_CLK in LPAV.

enumerator kCLOCK_LpavSysOscDiv2Clk
SOSCDIV2_CLK in LPAV.

enumerator kCLOCK_LpavSysOscDiv3Clk
SOSCDIV3_CLK in LPAV.

enumerator kCLOCK_Pll0Clk
CGC PLL0 clock. (PLL0CLK)

enumerator kCLOCK_Pll1Clk
CGC PLL1 clock. (PLL1CLK)

enumerator kCLOCK_Pll3Clk
CGC PLL3 clock. (PLL3CLK)

enumerator kCLOCK_Pll4Clk
CGC PLL4 clock. (PLL4CLK)

enumerator kCLOCK_Pll0Pfd0Clk
pll0 pfd0.

enumerator kCLOCK_Pll0Pfd1Clk
pll0 pfd1.

enumerator kCLOCK_Pll0Pfd2Clk
pll0 pfd2.

enumerator kCLOCK_Pll0Pfd3Clk
pll0 pfd3.

enumerator kCLOCK_Pll1Pfd0Clk
pll1 pfd0.

enumerator kCLOCK_Pll1Pfd1Clk
pll1 pfd1.

enumerator kCLOCK_Pll1Pfd2Clk
pll1 pfd2.

enumerator kCLOCK_Pll1Pfd3Clk
pll1 pfd3.

enumerator kCLOCK_Pll3Pfd0Clk
pll3 pfd0.

enumerator kCLOCK_Pll3Pfd1Clk
pll3 pfd1.

enumerator kCLOCK_Pll3Pfd2Clk
pll3 pfd2.

enumerator kCLOCK_Pll3Pfd3Clk
pll3 pfd3.

enumerator kCLOCK_Pll4Pfd0Clk
pll4 pfd0.

enumerator kCLOCK_Pll4Pfd1Clk
pll4 pfd1.

enumerator kCLOCK_Pll4Pfd2Clk
pll4 pfd2.

enumerator kCLOCK_Pll4Pfd3Clk
pll4 pfd3.

enumerator kCLOCK_Pll0VcoDivClk
PLL0VCODIV.

enumerator kCLOCK_Pll0Pfd1DivClk
PLL0PFD1DIV.

enumerator kCLOCK_Pll0Pfd2DivClk
PLL0PFD2DIV.

enumerator kCLOCK_Pll1VcoDivClk
PLL1VCODIV.

enumerator kCLOCK_Pll1Pfd1DivClk
PLL1PFD1DIV.

enumerator kCLOCK_Pll1Pfd2DivClk
PLL1PFD2DIV.

enumerator kCLOCK_Pll3VcoDivClk
PLL3VCODIV.

enumerator kCLOCK_Pll3Pfd0Div1Clk
PLL3PFD0DIV1.

enumerator kCLOCK_Pll3Pfd0Div2Clk
PLL3PFD0DIV2.

enumerator kCLOCK_Pll3Pfd1Div1Clk
PLL3PFD1DIV1.

enumerator kCLOCK_Pll3Pfd1Div2Clk
PLL3PFD1DIV2.

enumerator kCLOCK_Pll3Pfd2Div1Clk
PLL3PFD2DIV1.

enumerator kCLOCK_Pll3Pfd2Div2Clk
PLL3PFD2DIV2.

enumerator kCLOCK_Pll3Pfd3Div1Clk
PLL3PFD3DIV1.

enumerator kCLOCK_Pll3Pfd3Div2Clk
PLL3PFD3DIV2.

enumerator kCLOCK_Pll4VcoDivClk
PLL4VCODIV.

enumerator kCLOCK_Pll4Pfd0Div1Clk
PLL4PFD0DIV1.

enumerator kCLOCK_Pll4Pfd0Div2Clk
PLL4PFD0DIV2.

enumerator kCLOCK_Pll4Pfd1Div1Clk
PLL4PFD1DIV1.

enumerator kCLOCK_Pll4Pfd1Div2Clk
PLL4PFD1DIV2.

enumerator kCLOCK_Pll4Pfd2Div1Clk
PLL4PFD2DIV1.

enumerator kCLOCK_Pll4Pfd2Div2Clk
PLL4PFD2DIV2.

enumerator kCLOCK_Pll4Pfd3Div1Clk
PLL4PFD3DIV1.

enumerator kCLOCK_Pll4Pfd3Div2Clk
PLL4PFD3DIV2.

enum _clock_ip_src

Clock source for peripherals that support various clock selections.

Values:

enumerator kCLOCK_IpSrcNone

Clock is off.

enumerator kCLOCK_Pcc0PlatIpSrcSysOscDiv1

PCC0, PCC1 Platform clock selection: kCLOCK_RtdSysOscDiv1Clk

enumerator kCLOCK_Pcc0PlatIpSrcFroDiv1

PCC0, PCC1 Platform clock selection: kCLOCK_RtdFroDiv1Clk

enumerator kCLOCK_Pcc0PlatIpSrcCm33Plat

PCC0, PCC1 Platform clock selection: kCLOCK_Cm33CorePlatClk

enumerator kCLOCK_Pcc0PlatIpSrcFro

PCC0, PCC1 Platform clock selection: kCLOCK_FroClk

enumerator kCLOCK_Pcc0PlatIpSrcPll0Pfd3

PCC0, PCC1 Platform clock selection: kCLOCK_Pll0Pfd3Clk

enumerator kCLOCK_Pcc0BusIpSrcLpo

PCC0, PCC1 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc0BusIpSrcSysOscDiv2

PCC0, PCC1 Bus clock selection: kCLOCK_RtdSysOscDiv2Clk

enumerator kCLOCK_Pcc0BusIpSrcFroDiv2

PCC0, PCC1 Bus clock selection: kCLOCK_RtdFroDiv2Clk

enumerator kCLOCK_Pcc0BusIpSrcCm33Bus

PCC0, PCC1 Bus clock selection: kCLOCK_Cm33BusClk

enumerator kCLOCK_Pcc0BusIpSrcPll1Pfd1Div

PCC0 Bus clock selection: kCLOCK_Pll1Pfd1DivClk

enumerator kCLOCK_Pcc0BusIpSrcPll0Pfd2Div

PCC0, PCC1 Bus clock selection: kCLOCK_Pll0Pfd2DivClk

enumerator kCLOCK_Pcc0BusIpSrcPll0Pfd1Div

PCC0, PCC1 Bus clock selection: kCLOCK_Pll0Pfd1DivClk

enumerator kCLOCK_Pcc1PlatIpSrcSysOscDiv1

PCC0, PCC1 Platform clock selection: kCLOCK_RtdSysOscDiv1Clk

enumerator kCLOCK_Pcc1PlatIpSrcFroDiv1

PCC0, PCC1 Platform clock selection: kCLOCK_RtdFroDiv1Clk

enumerator kCLOCK_Pcc1PlatIpSrcCm33Plat

PCC0, PCC1 Platform clock selection: kCLOCK_Cm33CorePlatClk

enumerator kCLOCK_Pcc1PlatIpSrcFro

PCC0, PCC1 Platform clock selection: kCLOCK_FroClk

enumerator kCLOCK_Pcc1PlatIpSrcPll0Pfd3

PCC0, PCC1 Platform clock selection: kCLOCK_Pll0Pfd3Clk

enumerator kCLOCK_Pcc1BusIpSrcLpo

PCC0, PCC1 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc1BusIpSrcSysOscDiv2

PCC0, PCC1 Bus clock selection: kCLOCK_RtdSysOscDiv2Clk

enumerator kCLOCK_Pcc1BusIpSrcFroDiv2

PCC0, PCC1 Bus clock selection: kCLOCK_RtdFroDiv2Clk

enumerator kCLOCK_Pcc1BusIpSrcCm33Bus
PCC0, PCC1 Bus clock selection: kCLOCK_Cm33BusClk

enumerator kCLOCK_Pcc1BusIpSrcPll1VcoDiv
PCC1 Bus clock selection: kCLOCK_Pll1VcoDivClk

enumerator kCLOCK_Pcc1BusIpSrcPll0Pfd2Div
PCC0, PCC1 Bus clock selection: kCLOCK_Pll0Pfd2DivClk

enumerator kCLOCK_Pcc1BusIpSrcPll0Pfd1Div
PCC0, PCC1 Bus clock selection: kCLOCK_Pll0Pfd1DivClk

enumerator kCLOCK_Pcc2BusIpSrcLpo
PCC2 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc2BusIpSrcSysOscDiv3
PCC2 Bus clock selection: kCLOCK_RtdSysOscDiv3Clk

enumerator kCLOCK_Pcc2BusIpSrcFroDiv3
PCC2 Bus clock selection: kCLOCK_RtdFroDiv3Clk

enumerator kCLOCK_Pcc2BusIpSrcFusionDspBus
PCC2 Bus clock selection: kCLOCK_FusionDspBusClk

enumerator kCLOCK_Pcc2BusIpSrcPll1VcoDiv
PCC2 Bus clock selection: kCLOCK_Pll1VcoDivClk

enumerator kCLOCK_Pcc2BusIpSrcPll0Pfd2Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd2DivClk

enumerator kCLOCK_Pcc2BusIpSrcPll0Pfd1Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd1DivClk

enumerator kCLOCK_Pcc3BusIpSrcLpo
PCC3 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc3BusIpSrcSysOscDiv2
PCC3 Bus clock selection: kCLOCK_AdSysOscDiv2Clk

enumerator kCLOCK_Pcc3BusIpSrcFroDiv2
PCC3 Bus clock selection: kCLOCK_AdFroDiv2Clk

enumerator kCLOCK_Pcc3BusIpSrcXbarBus
PCC3 Bus clock selection: kCLOCK_XbarBusClk

enumerator kCLOCK_Pcc3BusIpSrcPll3Pfd1Div1
PCC3 Bus clock selection: kCLOCK_Pll3Pfd1Div1Clk

enumerator kCLOCK_Pcc3BusIpSrcPll3Pfd0Div2
PCC3 Bus clock selection: kCLOCK_Pll3Pfd0Div2Clk

enumerator kCLOCK_Pcc3BusIpSrcPll3Pfd0Div1
PCC3 Bus clock selection: kCLOCK_Pll3Pfd0Div1Clk

enumerator kCLOCK_Pcc4PlatIpSrcSysOscDiv1
PCC4 Platform clock selection: kCLOCK_AdSysOscDiv1Clk

enumerator kCLOCK_Pcc4PlatIpSrcFroDiv1
PCC4 Platform clock selection: kCLOCK_AdFroDiv1Clk

enumerator kCLOCK_Pcc4PlatIpSrcPll3Pfd3Div2
PCC4 Platform clock selection: kCLOCK_Pll3Pfd3Div2Clk

enumerator kCLOCK_Pcc4PlatIpSrcPll3Pfd3Div1
PCC4 Platform clock selection: kCLOCK_Pll3Pfd3Div1Clk

enumerator kCLOCK_Pcc4PlatIpSrcPll3Pfd2Div2
PCC4 Platform clock selection: kCLOCK_Pll3Pfd2Div2Clk

enumerator kCLOCK_Pcc4PlatIpSrcPll3Pfd2Div1
PCC4 Platform clock selection: kCLOCK_Pll3Pfd2Div1Clk

enumerator kCLOCK_Pcc4PlatIpSrcPll3Pfd1Div2
PCC4 Platform clock selection: kCLOCK_Pll3Pfd1Div2Clk

enumerator kCLOCK_Pcc4BusIpSrcLpo
PCC4 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc4BusIpSrcSysOscDiv2
PCC4 Bus clock selection: kCLOCK_AdSysOscDiv2Clk

enumerator kCLOCK_Pcc4BusIpSrcFroDiv2
PCC4 Bus clock selection: kCLOCK_AdFroDiv2Clk

enumerator kCLOCK_Pcc4BusIpSrcXbarBus
PCC4 Bus clock selection: kCLOCK_XbarBusClk

enumerator kCLOCK_Pcc4BusIpSrcPll3VcoDiv
PCC4 Bus clock selection: kCLOCK_Pll3VcoDivClk

enumerator kCLOCK_Pcc4BusIpSrcPll3Pfd0Div1
PCC4 Bus clock selection: kCLOCK_Pll3Pfd0Div1Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd3Div2
PCC5 Platform clock selection: kCLOCK_Pll4Pfd3Div2Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd2Div2
PCC5 Platform clock selection: kCLOCK_Pll4Pfd2Div2Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd2Div1
PCC5 Platform clock selection: kCLOCK_Pll4Pfd2Div1Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd1Div2
PCC5 Platform clock selection: kCLOCK_Pll4Pfd1Div2Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd1Div1
PCC5 Platform clock selection: kCLOCK_Pll4Pfd1Div1Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd0Div2
PCC5 Platform clock selection: kCLOCK_Pll4Pfd0Div2Clk

enumerator kCLOCK_Pcc5PlatIpSrcPll4Pfd0Div1
PCC5 Platform clock selection: kCLOCK_Pll4Pfd0Div1Clk

enumerator kCLOCK_Pcc5BusIpSrcLpo
PCC5 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_Pcc5BusIpSrcSysOscDiv2
PCC5 Bus clock selection: kCLOCK_LpavSysOscDiv2Clk

enumerator kCLOCK_Pcc5BusIpSrcFroDiv2
PCC5 Bus clock selection: kCLOCK_LpavFroDiv2Clk

enumerator kCLOCK_Pcc5BusIpSrcLpavBus
PCC5 Bus clock selection: kCLOCK_NicLpavBusClk

enumerator kCLOCK_Pcc5BusIpSrcPll4VcoDiv
PCC5 Bus clock selection: kCLOCK_Pll4VcoDivClk

enumerator kCLOCK_Pcc5BusIpSrcPll4Pfd3Div1
PCC5 Bus clock selection: kCLOCK_Pll4Pfd3Div1Clk

enumerator kCLOCK_Cm33SaiClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_Cm33SaiClkSrcRtdAudClk
Common audio clock in RTD, see cgc_rtd_audclk_src_t.

enumerator kCLOCK_Cm33SaiClkSrcLpavAudClk
Common audio clock in LPAV, see cgc_lpav_audclk_src_t.

enumerator kCLOCK_Cm33SaiClkSrcSysOsc
SYSOSC main reference clock to the chip: kCLOCK_SysOscClk.

enumerator kCLOCK_FusionSaiClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_FusionSaiClkSrcExtMelk1
External audio master clock input 1.

enumerator kCLOCK_FusionSaiClkSrcSai3Rx
SAI3 receiver serial bit clock. Only for SAI2.

enumerator kCLOCK_FusionSaiClkSrcSai2Tx
SAI2 transmitter serial bit clock. Only for SAI3.

enumerator kCLOCK_FusionSaiClkSrcSysOsc
SYSOSC main reference clock to the chip: kCLOCK_SysOscClk.

enumerator kCLOCK_FusionMicfilClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk

enumerator kCLOCK_FusionMicfilClkSrcFro24
FRO24: kCLOCK_FroClk/8.

enumerator kCLOCK_FusionMicfilClkSrcSysOsc
SYSOSC main reference clock to the chip: kCLOCK_SysOscClk.

enumerator kCLOCK_FusionMicfilClkSrcExtMelk1
External audio master clock input 1.

enumerator kCLOCK_FusionMicfilClkSrcRtcOsc
kCLOCK_RtcOscClk.

enumerator kCLOCK_FusionMicfilClkSrcLpo
kCLOCK_LpOscClk.

enumerator kCLOCK_FusionTpm2ClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_FusionTpm2ClkSrcExtMelk1
External audio master clock input 1.

enumerator kCLOCK_FusionTpm2ClkSrcLpo
PCC2 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_FusionTpm2ClkSrcSysOscDiv3
PCC2 Bus clock selection: kCLOCK_RtdSysOscDiv3Clk

enumerator kCLOCK_FusionTpm2ClkSrcFroDiv3
PCC2 Bus clock selection: kCLOCK_RtdFroDiv3Clk

enumerator kCLOCK_FusionTpm2ClkSrcFusionDspBus
PCC2 Bus clock selection: kCLOCK_FusionDspBusClk

enumerator kCLOCK_FusionTpm2ClkSrcPll1VcoDiv
PCC2 Bus clock selection: kCLOCK_Pll1VcoDivClk

enumerator kCLOCK_FusionTpm2ClkSrcPll0Pfd2Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd2DivClk

enumerator kCLOCK_FusionTpm2ClkSrcPll0Pfd1Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd1DivClk

enumerator kCLOCK_FusionTpm3ClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_FusionTpm3ClkSrcRtdAudClk
Common audio clock in RTD, see `cgc_rtd_audclk_src_t`.

enumerator kCLOCK_FusionTpm3ClkSrcLpavAudClk
Common audio clock in LPAV, see `cgc_lpav_audclk_src_t`.

enumerator kCLOCK_FusionTpm3ClkSrcLpo
PCC2 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_FusionTpm3ClkSrcSysOscDiv3
PCC2 Bus clock selection: kCLOCK_RtdSysOscDiv3Clk

enumerator kCLOCK_FusionTpm3ClkSrcFroDiv3
PCC2 Bus clock selection: kCLOCK_RtdFroDiv3Clk

enumerator kCLOCK_FusionTpm3ClkSrcFusionDspBus
PCC2 Bus clock selection: kCLOCK_FusionDspBusClk

enumerator kCLOCK_FusionTpm3ClkSrcPll1VcoDiv
PCC2 Bus clock selection: kCLOCK_Pll1VcoDivClk

enumerator kCLOCK_FusionTpm3ClkSrcPll0Pfd2Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd2DivClk

enumerator kCLOCK_FusionTpm3ClkSrcPll0Pfd1Div
PCC2 Bus clock selection: kCLOCK_Pll0Pfd1DivClk

enumerator kCLOCK_AdSaiClkSrcPll3Pfd1Div1
PLL3 PFD1 DIV1 in AD: kCLOCK_Pll3Pfd1Div2Clk.

enumerator kCLOCK_AdSaiClkSrcAdAudClk
Common audio clock in AD, see `cgc_ad_audclk_src_t`.

enumerator kCLOCK_AdSaiClkSrcLpavAudClk
Common audio clock in LPAV, see `cgc_lpav_audclk_src_t`.

enumerator kCLOCK_AdSaiClkSrcSysOsc
SYSOSC main reference clock to the chip: kCLOCK_SysOscClk.

enumerator kCLOCK_AdTpm67ClkSrcPll3Pfd1Div1
PLL3 PFD1 DIV1 in AD: kCLOCK_Pll3Pfd1Div2Clk.

enumerator kCLOCK_AdTpm67ClkSrcAdAudClk
Common audio clock in AD, see `cgc_ad_audclk_src_t`.

enumerator kCLOCK_AdTpm67ClkSrcLpavAudClk
Common audio clock in LPAV, see `cgc_lpav_audclk_src_t`.

enumerator kCLOCK_AdTpm67ClkSrcLpo
PCC4 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_AdTpm67ClkSrcSysOscDiv2
PCC4 Bus clock selection: kCLOCK_AdSysOscDiv2Clk

enumerator kCLOCK_AdTpm67ClkSrcFroDiv2
PCC4 Bus clock selection: kCLOCK_AdFroDiv2Clk

enumerator kCLOCK_AdTpm67ClkSrcXbarBus
PCC4 Bus clock selection: kCLOCK_XbarBusClk

enumerator kCLOCK_AdTpm67ClkSrcPll3VcoDiv
PCC4 Bus clock selection: kCLOCK_Pll3VcoDivClk

enumerator kCLOCK_AdTpm67ClkSrcPll3Pfd0Div1
PCC4 Bus clock selection: kCLOCK_Pll3Pfd0Div1Clk

enumerator kCLOCK_LpavSaiClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_LpavSaiClkSrcPll3Pfd1Div1
PLL3 PFD1 DIV1 in AD: kCLOCK_Pll3Pfd1Div1Clk.

enumerator kCLOCK_LpavSaiClkSrcRtdAudClk
Common audio clock in RTD, see `cgc_rtd_audclk_src_t`.

enumerator kCLOCK_LpavSaiClkSrcAdAudClk
Common audio clock in AD, see `cgc_ad_audclk_src_t`.

enumerator kCLOCK_LpavSaiClkSrcLpavAudClk
Common audio clock in LPAV, see `cgc_lpav_audclk_src_t`.

enumerator kCLOCK_LpavSaiClkSrcSysOsc
SYSOSC main reference clock to the chip.

enumerator kCLOCK_LpavTpm8ClkSrcPll1Pfd2Div
PLL1 PFD2 DIV in RTD: kCLOCK_Pll1Pfd2DivClk.

enumerator kCLOCK_LpavTpm8ClkSrcPll3Pfd1Div1
PLL3 PFD1 DIV1 in AD: kCLOCK_Pll3Pfd1Div1Clk.

enumerator kCLOCK_LpavTpm8ClkSrcRtdAudClk
Common audio clock in RTD, see `cgc_rtd_audclk_src_t`.

enumerator kCLOCK_LpavTpm8ClkSrcAdAudClk
Common audio clock in AD, see `cgc_ad_audclk_src_t`.

enumerator kCLOCK_LpavTpm8ClkSrcLpavAudClk
Common audio clock in LPAV, see `cgc_lpav_audclk_src_t`.

enumerator kCLOCK_LpavTpm8ClkSrcLpo
PCC5 Bus clock selection: kCLOCK_LpOscClk

enumerator kCLOCK_LpavTpm8ClkSrcSysOscDiv2
PCC5 Bus clock selection: kCLOCK_LpavSysOscDiv2Clk

enumerator kCLOCK_LpavTpm8ClkSrcFroDiv2
PCC5 Bus clock selection: kCLOCK_LpavFroDiv2Clk

enumerator kCLOCK_LpavTpm8ClkSrcLpavBus

PCC5 Bus clock selection: kCLOCK_NicLpavBusClk

enumerator kCLOCK_LpavTpm8ClkSrcPll4VcoDiv

PCC5 Bus clock selection: kCLOCK_Pll4VcoDivClk

enumerator kCLOCK_LpavTpm8ClkSrcPll4Pfd3Div1

PCC5 Bus clock selection: kCLOCK_Pll4Pfd3Div1Clk

enum _clock_lptmr_src

Clock source for LPTMR.

Values:

enumerator kCLOCK_LptmrSrcLPO1M

LPO 1MHz clock.

enumerator kCLOCK_LptmrSrcRtc1K

RTC 1KHz clock.

enumerator kCLOCK_LptmrSrcRtc32K

RTC 32KHz clock.

enumerator kCLOCK_LptmrSrcSysOsc

system OSC clock.

enum _clock_ip_name

Peripheral clock name definition used for clock gate, clock source and clock divider setting. [31:2] is defined as the corresponding register address. [1:1] is used as indicator of existing of PCS. [0:0] is used as indicator of existing of PCD/FAC.

Values:

enumerator kCLOCK_IpInvalid

enumerator kCLOCK_Dma0

enumerator kCLOCK_Dma0Ch0

enumerator kCLOCK_Dma0Ch1

enumerator kCLOCK_Dma0Ch2

enumerator kCLOCK_Dma0Ch3

enumerator kCLOCK_Dma0Ch4

enumerator kCLOCK_Dma0Ch5

enumerator kCLOCK_Dma0Ch6

enumerator kCLOCK_Dma0Ch7

enumerator kCLOCK_Dma0Ch8

enumerator kCLOCK_Dma0Ch9

enumerator kCLOCK_Dma0Ch10

enumerator kCLOCK_Dma0Ch11

enumerator kCLOCK_Dma0Ch12

enumerator kCLOCK_Dma0Ch13

enumerator kCLOCK__Dma0Ch14
enumerator kCLOCK__Dma0Ch15
enumerator kCLOCK__Dma0Ch16
enumerator kCLOCK__Dma0Ch17
enumerator kCLOCK__Dma0Ch18
enumerator kCLOCK__Dma0Ch19
enumerator kCLOCK__Dma0Ch20
enumerator kCLOCK__Dma0Ch21
enumerator kCLOCK__Dma0Ch22
enumerator kCLOCK__Dma0Ch23
enumerator kCLOCK__Dma0Ch24
enumerator kCLOCK__Dma0Ch25
enumerator kCLOCK__Dma0Ch26
enumerator kCLOCK__Dma0Ch27
enumerator kCLOCK__Dma0Ch28
enumerator kCLOCK__Dma0Ch29
enumerator kCLOCK__Dma0Ch30
enumerator kCLOCK__Dma0Ch31
enumerator kCLOCK__Mu0A
enumerator kCLOCK__Mu1A
enumerator kCLOCK__Mu2A
enumerator kCLOCK__Syspm0
enumerator kCLOCK__Wuu0
enumerator kCLOCK__UpowerMuARtd
enumerator kCLOCK__Wdog0
enumerator kCLOCK__Wdog1
enumerator kCLOCK__TrdcMgr
enumerator kCLOCK__TrdcMbc0
enumerator kCLOCK__TrdcMbc1
enumerator kCLOCK__TrdcMbc2
enumerator kCLOCK__TrdcMbc3
enumerator kCLOCK__TrdcMrc
enumerator kCLOCK__Sema420

enumerator kCLOCK_Bbnsn
enumerator kCLOCK_Flexspi0
enumerator kCLOCK_Lpit0
enumerator kCLOCK_Flexio0
enumerator kCLOCK_I3c0
enumerator kCLOCK_Lpspi0
enumerator kCLOCK_Lpspi1
enumerator kCLOCK_Adc0
enumerator kCLOCK_Cmp0
enumerator kCLOCK_Cmp1
enumerator kCLOCK_Dac0
enumerator kCLOCK_Dac1
enumerator kCLOCK_Cm33Cache
enumerator kCLOCK_SentinelLpuart
enumerator kCLOCK_PowersysWdog
enumerator kCLOCK_Ocotp
enumerator kCLOCK_Tpiu
enumerator kCLOCK_Swo
enumerator kCLOCK_Flexspi1
enumerator kCLOCK_Lptmr0
enumerator kCLOCK_Lptmr1
enumerator kCLOCK_Tpm0
enumerator kCLOCK_Tpm1
enumerator kCLOCK_Lpi2c0
enumerator kCLOCK_Lpi2c1
enumerator kCLOCK_Lpuart0
enumerator kCLOCK_Lpuart1
enumerator kCLOCK_Sai0
enumerator kCLOCK_Sai1
enumerator kCLOCK_DigFiltA
enumerator kCLOCK_DigFiltB
enumerator kCLOCK_Adc1
enumerator kCLOCK_Flexcan

enumerator kCLOCK_RgpioA
enumerator kCLOCK_RgpioB
enumerator kCLOCK_RgpioC
enumerator kCLOCK_Mu1B
enumerator kCLOCK_Wdog2
enumerator kCLOCK_Tpm2
enumerator kCLOCK_Tpm3
enumerator kCLOCK_Mrt
enumerator kCLOCK_Lpi2c2
enumerator kCLOCK_Lpi2c3
enumerator kCLOCK_I3c1
enumerator kCLOCK_Lpuart2
enumerator kCLOCK_Lpuart3
enumerator kCLOCK_Lpspi2
enumerator kCLOCK_Lpspi3
enumerator kCLOCK_Sai2
enumerator kCLOCK_Sai3
enumerator kCLOCK_Micfil
enumerator kCLOCK_Dma1
enumerator kCLOCK_Dma1Ch0
enumerator kCLOCK_Dma1Ch1
enumerator kCLOCK_Dma1Ch2
enumerator kCLOCK_Dma1Ch3
enumerator kCLOCK_Dma1Ch4
enumerator kCLOCK_Dma1Ch5
enumerator kCLOCK_Dma1Ch6
enumerator kCLOCK_Dma1Ch7
enumerator kCLOCK_Dma1Ch8
enumerator kCLOCK_Dma1Ch9
enumerator kCLOCK_Dma1Ch10
enumerator kCLOCK_Dma1Ch11
enumerator kCLOCK_Dma1Ch12
enumerator kCLOCK_Dma1Ch13

enumerator kCLOCK_Dma1Ch14
enumerator kCLOCK_Dma1Ch15
enumerator kCLOCK_Dma1Ch16
enumerator kCLOCK_Dma1Ch17
enumerator kCLOCK_Dma1Ch18
enumerator kCLOCK_Dma1Ch19
enumerator kCLOCK_Dma1Ch20
enumerator kCLOCK_Dma1Ch21
enumerator kCLOCK_Dma1Ch22
enumerator kCLOCK_Dma1Ch23
enumerator kCLOCK_Dma1Ch24
enumerator kCLOCK_Dma1Ch25
enumerator kCLOCK_Dma1Ch26
enumerator kCLOCK_Dma1Ch27
enumerator kCLOCK_Dma1Ch28
enumerator kCLOCK_Dma1Ch29
enumerator kCLOCK_Dma1Ch30
enumerator kCLOCK_Dma1Ch31
enumerator kCLOCK_Mu0B
enumerator kCLOCK_Mu3A
enumerator kCLOCK_Wuu1
enumerator kCLOCK_Syspm1
enumerator kCLOCK_UpowerMuAApd
enumerator kCLOCK_Wdog3
enumerator kCLOCK_Wdog4
enumerator kCLOCK_Caam
enumerator kCLOCK_XrdcMgr
enumerator kCLOCK_Sema421
enumerator kCLOCK_Lpit1
enumerator kCLOCK_Tpm4
enumerator kCLOCK_Tpm5
enumerator kCLOCK_Flexio1
enumerator kCLOCK_I3c2

enumerator kCLOCK_Lpi2c4
enumerator kCLOCK_Lpi2c5
enumerator kCLOCK_Lpuart4
enumerator kCLOCK_Lpuart5
enumerator kCLOCK_Lpspi4
enumerator kCLOCK_Lpspi5
enumerator kCLOCK_Flexspi2
enumerator kCLOCK_Tpm6
enumerator kCLOCK_Tpm7
enumerator kCLOCK_Lpi2c6
enumerator kCLOCK_Lpi2c7
enumerator kCLOCK_Lpuart6
enumerator kCLOCK_Lpuart7
enumerator kCLOCK_Sai4
enumerator kCLOCK_Sai5
enumerator kCLOCK_DigFiltE
enumerator kCLOCK_DigFiltF
enumerator kCLOCK_Usdhc0
enumerator kCLOCK_Usdhc1
enumerator kCLOCK_Usdhc2
enumerator kCLOCK_Usb0
enumerator kCLOCK_Usb0Phy
enumerator kCLOCK_Usb1
enumerator kCLOCK_Usb1Phy
enumerator kCLOCK_UsbXbar
enumerator kCLOCK_Enet
enumerator kCLOCK_RgpioE
enumerator kCLOCK_RgpioF
enumerator kCLOCK_Dma2
enumerator kCLOCK_Dma2Ch0
enumerator kCLOCK_Dma2Ch1
enumerator kCLOCK_Dma2Ch2
enumerator kCLOCK_Dma2Ch3

enumerator kCLOCK_Dma2Ch4
enumerator kCLOCK_Dma2Ch5
enumerator kCLOCK_Dma2Ch6
enumerator kCLOCK_Dma2Ch7
enumerator kCLOCK_Dma2Ch8
enumerator kCLOCK_Dma2Ch9
enumerator kCLOCK_Dma2Ch10
enumerator kCLOCK_Dma2Ch11
enumerator kCLOCK_Dma2Ch12
enumerator kCLOCK_Dma2Ch13
enumerator kCLOCK_Dma2Ch14
enumerator kCLOCK_Dma2Ch15
enumerator kCLOCK_Dma2Ch16
enumerator kCLOCK_Dma2Ch17
enumerator kCLOCK_Dma2Ch18
enumerator kCLOCK_Dma2Ch19
enumerator kCLOCK_Dma2Ch20
enumerator kCLOCK_Dma2Ch21
enumerator kCLOCK_Dma2Ch22
enumerator kCLOCK_Dma2Ch23
enumerator kCLOCK_Dma2Ch24
enumerator kCLOCK_Dma2Ch25
enumerator kCLOCK_Dma2Ch26
enumerator kCLOCK_Dma2Ch27
enumerator kCLOCK_Dma2Ch28
enumerator kCLOCK_Dma2Ch29
enumerator kCLOCK_Dma2Ch30
enumerator kCLOCK_Dma2Ch31
enumerator kCLOCK_Mu2B
enumerator kCLOCK_Mu3B
enumerator kCLOCK_Sema422
enumerator kCLOCK_Tpm8
enumerator kCLOCK_Sai6

enumerator kCLOCK__Sai7
enumerator kCLOCK__Spdif
enumerator kCLOCK__Isi
enumerator kCLOCK__CsiRegs
enumerator kCLOCK__Csi
enumerator kCLOCK__Dsi
enumerator kCLOCK__Wdog5
enumerator kCLOCK__Epdc
enumerator kCLOCK__Pxp
enumerator kCLOCK__Gpu2d
enumerator kCLOCK__Gpu3d
enumerator kCLOCK__Dcnano
enumerator kCLOCK__Lpddr4
enumerator kCLOCK__CsiClkUi
enumerator kCLOCK__CsiClkEsc
enumerator kCLOCK__RgpioD
enumerator kCLOCK__Mqs0
enumerator kCLOCK__Mqs1

CGC status return codes.

Values:

enumerator kStatus_CGC_Busy
Clock is busy.
enumerator kStatus_CGC_InvalidSrc
Invalid source.

enum _cgc_sys_clk

CGC system clock type.

Values:

enumerator kCGC_SysClkSlow
System slow clock.
enumerator kCGC_SysClkBus
Bus clock.
enumerator kCGC_SysClkCorePlat
Core/Platform clock.
enumerator kCGC_SysClkHifi4
Hifi4 core clock.

enumerator kCGC_SysClkNicHifi
Hifi4 platform clock.

enumerator kCGC_SysClkLpavAxi
LPAV AXI clock.

enumerator kCGC_SysClkLpavAhb
LPAV AHB clock.

enumerator kCGC_SysClkLpavBus
LPAV Bus clock.

enum _cgc_rtd_sys_clk_src
CGC system clock source for RTD.

Values:

enumerator kCGC_RtdSysClkSrcFro
FRO192.

enumerator kCGC_RtdSysClkSrcPll0Pfd0
PLL0 PFD0.

enumerator kCGC_RtdSysClkSrcPll1Pfd0
PLL1 PFD0.

enumerator kCGC_RtdSysClkSrcSysOsc
System OSC.

enumerator kCGC_RtdSysClkSrcRtcOsc
RTC OSC.

enumerator kCGC_RtdSysClkSrcLvds
LVDS XCVR.

enumerator kCGC_RtdSysClkSrcPll0
PLL0.

enum _cgc_nic_sys_clk_src
CGC system clock source for NIC in AD.

Values:

enumerator kCGC_NicSysClkSrcFro
FRO192.

enumerator kCGC_NicSysClkSrcPll3Pfd0
PLL0 PFD0.

enumerator kCGC_NicSysClkSrcSysOsc
System OSC.

enumerator kCGC_NicSysClkSrcLvds
LVDS XCVR.

enum _cgc_hifi_sys_clk_src
CGC system clock source for HIFI4 in LPAV.

Values:

enumerator kCGC_HifiSysClkSrcFro
FRO192.

enumerator kCGC__HifiSysClkSrcPll4
PLL4.

enumerator kCGC__HifiSysClkSrcPll4Pfd0
PLL4 PFD0.

enumerator kCGC__HifiSysClkSrcSysOsc
System OSC.

enumerator kCGC__HifiSysClkSrcLvds
LVDS XCVR.

enum __cgc_lpav_sys_clk_src
CGC system clock source for LPAV.

Values:

enumerator kCGC__LpavSysClkSrcFro
FRO192.

enumerator kCGC__LpavSysClkSrcPll4Pfd1
PLL4 PFD1.

enumerator kCGC__LpavSysClkSrcSysOsc
System OSC.

enumerator kCGC__LpavSysClkSrcLvds
LVDS XCVR.

enum __cgc_ddr_sys_clk_src
CGC system clock source for DDR.

Values:

enumerator kCGC__DdrSysClkSrcFro
FRO192.

enumerator kCGC__DdrSysClkSrcPll4Pfd1
PLL4 PFD1.

enumerator kCGC__DdrSysClkSrcSysOsc
System OSC.

enumerator kCGC__DdrSysClkSrcLvds
LVDS XCVR.

enum __clock_rtd_clkout_src
CGC clock out configuration (CLKOUTCFG) in RTD.

Values:

enumerator kClockRtdClkoutSelCm33Core
CGC CM33 Core/Platform clock.

enumerator kClockRtdClkoutSelCm33Bus
CGC CM33 Bus clock.

enumerator kClockRtdClkoutSelCm33Slow
CGC CM33 Slow clock.

enumerator kClockRtdClkoutSelFusionDspCore
CGC Fusion DSP Core/Platform clock.

enumerator kClockRtdClkoutSelFusionDspBus
CGC Fusion DSP Bus clock.

enumerator kClockRtdClkoutSelFusionDspSlow
CGC Fusion DSP Slow clock.

enumerator kClockRtdClkoutSelFro48
FRO48: kCLOCK_FroClk/4.

enumerator kClockRtdClkoutSelPll0VcoDiv
PLL0 VCO DIV: kCLOCK_Pll0VcoDivClk.

enumerator kClockRtdClkoutSelPll1VcoDiv
PLL1 VCO DIV: kCLOCK_Pll1VcoDivClk.

enumerator kClockRtdClkoutSelSysOsc
System OSC.

enumerator kClockRtdClkoutSelLpOsc
CGC LPOSC clock.

enum _clock_lpav_clkout_src
CGC clock out configuration (CLKOUTCFG) in LPAV.

Values:

enumerator kClockLpavClkoutSelHifi4
CGC HIFI4 core clock.

enumerator kClockLpavClkoutSelNicHifi
CGC HIFI4 platform clock.

enumerator kClockLpavClkoutSelLpavAxi
CGC NIC LPAV AXI clock.

enumerator kClockLpavClkoutSelLpavAhb
CGC NIC LPAV AHB clock.

enumerator kClockLpavClkoutSelLpavBus
CGC LPAV Bus clock.

enumerator kClockLpavClkoutSelDdr
CGC DDR clock.

enumerator kClockLpavClkoutSelFro48
FRO48: kCLOCK_FroClk/4.

enumerator kClockLpavClkoutSelPll4VcoDiv
PLL4 VCO DIV: kCLOCK_Pll4VcoDivClk.

enumerator kClockLpavClkoutSelSysOsc
System OSC.

enumerator kClockLpavClkoutSelLpOsc
CGC LPOSC clock.

enum _cgc_async_clk
CGC asynchronous clock type.

Values:

enumerator kCGC_AsyncDiv1Clk
The async clock by DIV1, e.g. SOSCDIV1_CLK, FRODIV1_CLK.

enumerator kCGC__AsyncDiv2Clk

The async clock by DIV2, e.g. SOSCDIV2_CLK, FRODIV2_CLK.

enumerator kCGC__AsyncDiv3Clk

The async clock by DIV3, e.g. SOSCDIV3_CLK, FRODIV3_CLK.

enumerator kCGC__AsyncVcoClk

The async clock by PLL VCO DIV.

enumerator kCGC__AsyncPfd0Div1Clk

The async clock by PLL PFD0 DIV1.

enumerator kCGC__AsyncPfd0Div2Clk

The async clock by PLL PFD0 DIV2.

enumerator kCGC__AsyncPfd1Div1Clk

The async clock by PLL PFD1 DIV or DIV1.

enumerator kCGC__AsyncPfd1Div2Clk

The async clock by PLL PFD1 DIV2.

enumerator kCGC__AsyncPfd2Div1Clk

The async clock by PLL PFD2 DIV or DIV1.

enumerator kCGC__AsyncPfd2Div2Clk

The async clock by PLL PFD2 DIV2.

enumerator kCGC__AsyncPfd3Div1Clk

The async clock by PLL PFD3 DIV1.

enumerator kCGC__AsyncPfd3Div2Clk

The async clock by PLL PFD3 DIV2.

enum _cgc_sosc_monitor_mode

CGC system OSC monitor mode.

Values:

enumerator kCGC__SysOscMonitorDisable

Monitor disabled.

enumerator kCGC__SysOscMonitorInt

Interrupt when the system OSC error is detected.

enumerator kCGC__SysOscMonitorReset

Reset when the system OSC error is detected.

enum _cgc_sosc_mode

OSC work mode.

Values:

enumerator kCGC__SysOscModeExt

Use external clock.

enumerator kCGC__SysOscModeOscLowPower

Oscillator low power.

enumerator kCGC__SysOscModeOscHighGain

Oscillator high gain.

enum _cgc_sosc_enable_mode

OSC enable mode.

Values:

enumerator kCGC_SysOscEnableInDeepSleep

Enable OSC in deep sleep mode.

enumerator kCGC_SysOscEnableInPowerDown

Enable OSC in low power mode.

enum _cgc_fro_enable_mode

FRO enable mode.

Values:

enumerator kCGC_FroEnableInDeepSleep

Enable FRO in deep sleep mode.

enum _cgc_lposc_enable_mode

LPOSC enable mode.

Values:

enumerator kCGC_LposcEnableInDeepSleep

Enable OSC in deep sleep mode.

enumerator kCGC_LposcEnableInPowerDown

Enable OSC in low power mode.

enum _cgc_pll_src

CGC PLL clock source.

Values:

enumerator kCGC_PllSrcSysOsc

PLL clock source is system OSC.

enumerator kCGC_PllSrcFro24M

PLL clock source is FRO 24M.

enum _cgc_pll_enable_mode

PLL enable mode.

Values:

enumerator kCGC_PllEnable

Enable PLL clock.

enumerator kCGC_PllEnableInDeepSleep

Enable PLL in deep sleep mode.

enum _cgc_pll_pfd_clkout

CGC PLL PFD clouk out select.

Values:

enumerator kCGC_PllPfd0Clk

PFD0 output clock selected.

enumerator kCGC_PllPfd1Clk

PFD1 output clock selected.

enumerator kCGC_PllPfd2Clk

PFD2 output clock selected.

enumerator kCGC_PllPfd3Clk

PFD3 output clock selected.

enum _cgc_pll0_mult

PLL0 Multiplication Factor.

Values:

enumerator kCGC_Pll0Mult15

Divide by 15

enumerator kCGC_Pll0Mult16

Divide by 16

enumerator kCGC_Pll0Mult20

Divide by 20

enumerator kCGC_Pll0Mult22

Divide by 22

enumerator kCGC_Pll0Mult25

Divide by 25

enumerator kCGC_Pll0Mult30

Divide by 30

enum _cgc_rosc_monitor_mode

CGC RTC OSC monitor mode.

Values:

enumerator kCGC_RtcOscMonitorDisable

Monitor disabled.

enumerator kCGC_RtcOscMonitorInt

Interrupt when the RTC OSC error is detected.

enumerator kCGC_RtcOscMonitorReset

Reset when the RTC OSC error is detected.

enum _cgc_pll1_mult

PLL1 Multiplication Factor.

Values:

enumerator kCGC_Pll1Mult16

Divide by 16

enumerator kCGC_Pll1Mult17

Divide by 17

enumerator kCGC_Pll1Mult20

Divide by 20

enumerator kCGC_Pll1Mult22

Divide by 22

enumerator kCGC_Pll1Mult27

Divide by 27

enumerator kCGC_Pll1Mult33

Divide by 33

enum _cgc_pll4_mult

PLL4 Multiplication Factor.

Values:

enumerator kCGC_Pll4Mult16

Divide by 16

enumerator kCGC_Pll4Mult17

Divide by 17

enumerator kCGC_Pll4Mult20

Divide by 20

enumerator kCGC_Pll4Mult22

Divide by 22

enumerator kCGC_Pll4Mult27

Divide by 27

enumerator kCGC_Pll4Mult33

Divide by 33

enum _cgc_rtd_audclk_src

AUD_CLK0 source in RTD.

Values:

enumerator kCGC_RtdAudClkSrcExtMclk0

External audio master clock input 0 from pin.

enumerator kCGC_RtdAudClkSrcExtMclk1

External audio master clock input 1 from pin.

enumerator kCGC_RtdAudClkSrcSai0RxBclk

SAI0 receiver serial bit clock.

enumerator kCGC_RtdAudClkSrcSai0TxBclk

SAI0 transmitter serial bit clock.

enumerator kCGC_RtdAudClkSrcSai1RxBclk

SAI1 receiver serial bit clock.

enumerator kCGC_RtdAudClkSrcSai1TxBclk

SAI1 transmitter serial bit clock.

enumerator kCGC_RtdAudClkSrcSai2RxBclk

SAI2 receiver serial bit clock.

enumerator kCGC_RtdAudClkSrcSai2TxBclk

SAI2 transmitter serial bit clock.

enumerator kCGC_RtdAudClkSrcSai3RxBclk

SAI3 receiver serial bit clock.

enumerator kCGC_RtdAudClkSrcSai3TxBclk

SAI3 transmitter serial bit clock.

enum _cgc_ad_audclk_src

AUD_CLK1 source in AD.

Values:

enumerator kCGC_AdAudClkSrcExtMclk2

External audio master clock input 2 from pin.

enumerator kCGC_AdAudClkSrcSai4RxBclk

SAI4 receiver serial bit clock.

enumerator kCGC_AdAudClkSrcSai4TxBclk
SAI4 transmitter serial bit clock.

enumerator kCGC_AdAudClkSrcSai5RxBclk
SAI5 receiver serial bit clock.

enumerator kCGC_AdAudClkSrcSai5TxBclk
SAI5 transmitter serial bit clock.

enum _cgc_lpav_audclk_src
AUD_CLK2 source in LPAV.

Values:

enumerator kCGC_LpavAudClkSrcExtMclk3
External audio master clock input 3 from pin.

enumerator kCGC_LpavAudClkSrcSai6RxBclk
SAI6 receiver serial bit clock.

enumerator kCGC_LpavAudClkSrcSai6TxBclk
SAI6 transmitter serial bit clock.

enumerator kCGC_LpavAudClkSrcSai7RxBclk
SAI7 receiver serial bit clock.

enumerator kCGC_LpavAudClkSrcSai7TxBclk
SAI7 transmitter serial bit clock.

enumerator kCGC_LpavAudClkSrcSpdifRx
SPDIF receiver clock.

typedef enum _clock_name clock_name_t
Clock name used to get clock frequency.

typedef enum _clock_ip_src clock_ip_src_t
Clock source for peripherals that support various clock selections.

typedef enum _clock_lptmr_src clock_lptmr_src_t
Clock source for LPTMR.

typedef enum _clock_ip_name clock_ip_name_t
Peripheral clock name definition used for clock gate, clock source and clock divider setting. [31:2] is defined as the corresponding register address. [1:1] is used as indicator of existing of PCS. [0:0] is used as indicator of existing of PCD/Frac.

typedef enum _cgc_sys_clk cgc_sys_clk_t
CGC system clock type.

typedef enum _cgc_rtd_sys_clk_src cgc_rtd_sys_clk_src_t
CGC system clock source for RTD.

typedef enum _cgc_nic_sys_clk_src cgc_nic_sys_clk_src_t
CGC system clock source for NIC in AD.

typedef enum _cgc_hifi_sys_clk_src cgc_hifi_sys_clk_src_t
CGC system clock source for HIFI4 in LPAV.

typedef enum _cgc_lpav_sys_clk_src cgc_lpav_sys_clk_src_t
CGC system clock source for LPAV.

typedef enum _cgc_ddr_sys_clk_src cgc_ddr_sys_clk_src_t
CGC system clock source for DDR.

```
typedef struct _cgc_rtd_sys_clk_config cgc_rtd_sys_clk_config_t
    CGC system clock configuration for RTD.

typedef struct _cgc_hifi_sys_clk_config cgc_hifi_sys_clk_config_t
    CGC system clock configuration for HIFI4 DSP.

typedef struct _cgc_lpav_sys_clk_config cgc_lpav_sys_clk_config_t
    CGC system clock configuration for LPAV.

typedef struct _cgc_ddr_sys_clk_config cgc_ddr_sys_clk_config_t
    CGC system clock configuration for DDR in LPAV.

typedef enum _clock_rtd_clkout_src clock_rtd_clkout_src_t
    CGC clock out configuration (CLKOUTCFG) in RTD.

typedef enum _clock_lpav_clkout_src clock_lpav_clkout_src_t
    CGC clock out configuration (CLKOUTCFG) in LPAV.

typedef enum _cgc_async_clk cgc_async_clk_t
    CGC asynchronous clock type.

typedef enum _cgc_sosc_monitor_mode cgc_sosc_monitor_mode_t
    CGC system OSC monitor mode.

typedef enum _cgc_sosc_mode cgc_sosc_mode_t
    OSC work mode.

typedef struct _cgc_sosc_config cgc_sosc_config_t
    CGC system OSC configuration.

typedef struct _cgc_fro_config cgc_fro_config_t
    CGC FRO clock configuration.

typedef struct _cgc_lposc_config cgc_lposc_config_t
    CGC LPOSC clock configuration.

typedef enum _cgc_pll_src cgc_pll_src_t
    CGC PLL clock source.

typedef enum _cgc_pll_pfd_clkout cgc_pll_pfd_clkout_t
    CGC PLL PFD clouk out select.

typedef enum _cgc_pll0_mult cgc_pll0_mult_t
    PLL0 Multiplication Factor.

typedef struct _cgc_pll0_config cgc_pll0_config_t
    CGC PLL0 configuration.

typedef enum _cgc_rosc_monitor_mode cgc_rosc_monitor_mode_t
    CGC RTC OSC monitor mode.

typedef struct _cgc_rosc_config cgc_rosc_config_t
    CGC RTC OSC configuration.

typedef enum _cgc_pll1_mult cgc_pll1_mult_t
    PLL1 Multiplication Factor.

typedef struct _cgc_pll1_config cgc_pll1_config_t
    CGC PLL1 configuration.

typedef enum _cgc_pll4_mult cgc_pll4_mult_t
    PLL4 Multiplication Factor.
```

`typedef struct _cgc_pll4_config cgc_pll4_config_t`
CGC PLL4 configuration.

`typedef enum _cgc_rtd_audclk_src cgc_rtd_audclk_src_t`
AUD_CLK0 source in RTD.

`typedef enum _cgc_ad_audclk_src cgc_ad_audclk_src_t`
AUD_CLK1 source in AD.

`typedef enum _cgc_lpav_audclk_src cgc_lpav_audclk_src_t`
AUD_CLK2 source in LPAV.

`volatile uint32_t g_xtal0Freq`
External XTAL (SYSOSC) clock frequency.

The XTAL (SYSOSC) clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetXtalFreq` to set the value in the clock driver. For example, if XTAL is 24 MHz:

```
CLOCK_InitSysOsc(...);  
CLOCK_SetXtalFreq(24000000);
```

This is important for the multicore platforms where only one core needs to set up the OSC/SYSOSC using `CLOCK_InitSysOsc`. All other cores need to call the `CLOCK_SetXtalFreq` to get a valid clock frequency.

`volatile uint32_t g_xtal32Freq`
External XTAL32/EXTAL32 clock frequency.

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

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

`volatile uint32_t g_lvdsFreq`
External LVDS pad clock frequency.

The LVDS pad clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetLvdsFreq` to set the value in the clock driver.

`volatile uint32_t g_mclkFreq[4]`
External MCLK pad clock frequency.

The MCLK pad clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetMclkFreq` to set the value in the clock driver.

`volatile uint32_t g_rxBclkFreq[8]`
External RX_BCLK pad clock frequency.

The RX_BCLK pad clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetRxBclkFreq` to set the value in the clock driver.

`volatile uint32_t g_txBclkFreq[8]`
External TX_BCLK pad clock frequency.

The TX_BCLK pad clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetTxBclkFreq` to set the value in the clock driver.

`volatile uint32_t g_spdifRxFreq`
Recovered SPDIF_RX clock frequency.

The SPDIF_RX clock frequency in Hz. When the clock is sampled, use the function `CLOCK_SetSpdifRxFreq` to set the value in the clock driver.

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 bool CLOCK_IsEnabledByOtherCore(*clock_ip_name_t* name)

Check whether the clock is already enabled and configured by any other core.

Parameters

- name – Which peripheral to check, see *clock_ip_name_t*.

Returns

True if clock is already enabled, otherwise false.

void CLOCK_SetIpSrc(*clock_ip_name_t* name, *clock_ip_src_t* src)

Set the clock source for specific IP module.

Set the clock source for specific IP, not all modules need to set the clock source, should only use this function for the modules need source setting.

Parameters

- name – Which peripheral to check, see *clock_ip_name_t*.
- src – Clock source to set.

void CLOCK_SetIpSrcDiv(*clock_ip_name_t* name, *clock_ip_src_t* src, uint8_t divValue, uint8_t fracValue)

Set the clock source and divider for specific IP module.

Set the clock source and divider for specific IP, not all modules need to set the clock source and divider, should only use this function for the modules need source and divider setting.

Divider output clock = Divider input clock x [(fracValue+1)/(divValue+1)].

Parameters

- name – Which peripheral to check, see *clock_ip_name_t*.
- src – Clock source to set.
- divValue – The divider value.
- fracValue – The fraction multiply value.

static inline void CLOCK_SetRtdAudClkSrc(*cgc_rtd_audclk_src_t* src)

Set the AUD_CLK0 source in RTD. NOTE: The audio clock pin frequency is decided by SAI, but clock driver cannot depend on SAI driver to get its pin frequencies, user need to explicitly set the MCLK/BCLK frequencies if other modules use the AUD_CLK0 source and need to get correct frequency.

Parameters

- src – Clock source to set.

static inline void CLOCK_SetAdAudClkSrc(*cgc_ad_audclk_src_t* src)

Set the AUD_CLK1 source in AD. NOTE: The audio clock pin frequency is decided by SAI, but clock driver cannot depend on SAI driver to get its pin frequencies, user need to explicitly set the MCLK/BCLK frequencies if other modules use the AUD_CLK1 source and need to get correct frequency.

Parameters

- `src` – Clock source to set.

static inline void CLOCK_SetLpavAudClkSrc(*cgc_lpav_audclk_src_t* src)

Set the AUD_CLK2 source in LPAV. NOTE: The audio clock pin frequency is decided by SAI, but clock driver cannot depend on SAI driver to get its pin frequencies, user need to explicitly set the MCLK/BCLK frequencies if other modules use the AUD_CLK2 source and need to get correct frequency.

Parameters

- `src` – Clock source to set.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in *clock_name_t*.

Parameters

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

Returns

Clock frequency value in hertz

uint32_t CLOCK_GetCm33CorePlatClkFreq(void)

Get the CM33 core/platform clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetCm33BusClkFreq(void)

Get the CM33 bus clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetCm33SlowClkFreq(void)

Get the CM33 slow clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFusionDspCorePlatClkFreq(void)

Get the Fusion DSP core/platform clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFusionDspBusClkFreq(void)

Get the Fusion DSP bus clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFusionDspSlowClkFreq(void)

Get the Fusion DSP slow clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetLvdsClkFreq(void)

Get the external LVDS pad clock frequency (LVDS).

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetIpFreq(*clock_ip_name_t* name)

Gets the clock frequency for a specific IP module.

This function gets the IP module clock frequency. It is only used for the IP modules which could select clock source by CLOCK_SetIpSrc().

Parameters

- name – Which peripheral to get, see clock_ip_name_t.

Returns

Clock frequency value in hertz

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.0.5.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

CGC_PLLPFD_PFD_VAL(pfdClkout, fracValue)

CGC (A/S)PLL PFD[PFDx] value.

CGC_PLLPFD_PFD_MASK(pfdClkout)

CGC (A/S)PLL PFD[PFD] mask.

CGC_PLLPFD_PFD_VALID_MASK(pfdClkout)

CGC (A/S)PLL PFD[PFDx_VALID] mask.

CGC_PLLPFD_PFD_CLKGATE_MASK(pfdClkout)

CGC (A/S)PLL PFD[PFDx_CLKGATE] mask.

PCC_CLKCFG_PCD_MASK

Re-define PCC register masks and bitfield operations to unify the different namings in the soc header file.

PCC_CLKCFG_PCD_SHIFT

PCC_CLKCFG_PCD(x)

PCC_CLKCFG_FRAC_MASK

PCC_CLKCFG_FRAC_SHIFT

PCC_CLKCFG_FRAC(x)

PCC_CLKCFG_PCS_MASK

PCC_CLKCFG_PCS_SHIFT

PCC_CLKCFG_PCS(x)

PCC_CLKCFG_INUSE_MASK

PCC_CLKCFG_CGC_MASK

PCC_CLKCFG_PR_MASK

PCC_PCS_VAL(reg)

Bitfield values for general PCC registers.

PCC_FRAC_VAL(reg)

PCC_PCD_VAL(reg)

CLOCK_IP_SOURCE_PCC_INDEX(idx)

Clock source index macros for clock_ip_src_t.

CLOCK_IP_SOURCE_NON_PCC_INDEX(idx)
CLOCK_IP_SOURCE_PCC_INDEX_VAL(src)
CLOCK_IP_SOURCE_NON_PCC_INDEX_VAL(src)
PCC_PCS_AVAIL_MASK
 Define PCC bit available mask for clock_ip_name_t.
PCC_PCD_FRAC_AVAIL_MASK
IP_NAME_NON_PCC_FLAG_MASK
 Define Non-PCC register flag mask for clock_ip_name_t.
PCC_REG(name)
 Define PCC register content for clock_ip_name_t.
RGPIO_CLOCKS
 Clock ip name array for RGPIO2P.
SAI_CLOCKS
 Clock ip name array for SAI.
PCTL_CLOCKS
 Clock ip name array for PCTL.
LPI2C_CLOCKS
 Clock ip name array for LPI2C.
I3C_CLOCKS
 Clock ip name array for I3C.
FLEXIO_CLOCKS
 Clock ip name array for FLEXIO.
FLEXCAN_CLOCKS
 Clock ip name array for FLEXCAN.
PDM_CLOCKS
 Clock ip name array for PDM.
LCDIF_CLOCKS
 Clock ip name array for LCDIF/DCNANO.
MIPI_DSI_HOST_CLOCKS
 Clock ip name array for MIPI DSI.
EDMA_CLOCKS
 Clock ip name array for EDMA.
EDMA_CHAN_CLOCKS
 Clock ip name array for EDMA Channels.
LPUART_CLOCKS
 Clock ip name array for LPUART.
DAC_CLOCKS
 Clock ip name array for DAC.
LPTMR_CLOCKS
 Clock ip name array for LPTMR.

LPADC_CLOCKS

Clock ip name array for LPADC.

LPSPI_CLOCKS

Clock ip name array for LPSPI.

TPM_CLOCKS

Clock ip name array for TPM.

LPIT_CLOCKS

Clock ip name array for LPIT.

CMP_CLOCKS

Clock ip name array for CMP.

WDOG_CLOCKS

Clock ip name array for MU.

Clock ip name array for WDOG.

SEMA42_CLOCKS

Clock ip name array for SEMA42.

TPIU_CLOCKS

Clock ip name array for TPIU.

FLEXSPI_CLOCKS

Clock ip name array for QSPI.

MRT_CLOCKS

Clock ip name array for MRT.

BBNSM_CLOCKS

Clock ip name array for BBNSM.

PXP_CLOCKS

Clock ip name array for PXP.

EPDC_CLOCKS

Clock ip name array for EPDC.

uint32_t CLOCK_GetCm33SysClkFreq(*cgc_sys_clk_t* type)

Gets the CGC CM33 system clock frequency.

This function gets the CGC CM33 system clock frequency. These clocks are used for core, platform, bus and slow clock domains.

Parameters

- type – Which type of clock to get.

Returns

Clock frequency.

static inline void CLOCK_SetCm33SysClkConfig(*cgc_rtd_sys_clk_config_t* *config)

Sets the system clock configuration for CM33 domain.

This function sets the system clock configuration for CM33 domain.

Parameters

- config – Pointer to the configuration.

uint32_t CLOCK_GetFusionDspSysClkFreq(*cgc_sys_clk_t* type)

Gets the CGC Fusion DSP system clock frequency.

This function gets the CGC Fusion DSP system clock frequency. These clocks are used for core, platform, bus and slow clock domains.

Parameters

- type – Which type of clock to get.

Returns

Clock frequency.

static inline void CLOCK_SetFusionSysClkConfig(const *cgc_rtd_sys_clk_config_t* *config)

Sets the system clock configuration for FusionF1 DSP domain.

This function sets the system clock configuration for FusionF1 DSP domain.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetCm33SysClkConfig(*cgc_rtd_sys_clk_config_t* *config)

Gets the system clock configuration for CM33 domain.

This function gets the system configuration for CM33 domain.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetFusionDspSysClkConfig(*cgc_rtd_sys_clk_config_t* *config)

Gets the system clock configuration for FusionF1 DSP domain.

This function gets the system configuration for FusionF1 DSP domain.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_SetRtdClkOutConfig(*clock_rtd_clkout_src_t* setting, uint8_t div, bool enable)

Sets the clock out configuration in RTD.

This function sets the clock out configuration.

Parameters

- setting – The selection to set.
- div – The divider to set (div > 0).
- enable – Enable clock out.

static inline void CLOCK_SetRtcClkOutConfig(uint8_t div)

Sets the RTC_CLOCKOUT configuration.

This function sets the RTC_CLOCKOUT configuration.

Parameters

- div – The divider to set (div > 0).

uint32_t CLOCK_GetXbarBusClkFreq(void)

Gets the CGC XBAR bus clock frequency in AD.

This function gets the CGC XBAR bus clock frequency.

Returns

Clock frequency.

uint32_t CLOCK_GetHifiDspSysClkFreq(*cgc_sys_clk_t* type)

Gets the CGC HIFI DSP system clock frequency in LPAV.

This function gets the CGC HIFI DSP system clock frequency. These clocks are used for core, platform domains.

Parameters

- type – Which type of clock to get.

Returns

Clock frequency.

static inline void CLOCK_SetHifiDspSysClkConfig(const *cgc_hifi_sys_clk_config_t* *config)

Sets the system clock configuration for HIFI4 DSP domain.

This function sets the system clock configuration for HIFI4 DSP domain.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetHifiDspSysClkConfig(*cgc_hifi_sys_clk_config_t* *config)

Gets the system clock configuration for HIFI4 DSP domain.

This function gets the system configuration for HIFI4 DSP domain.

Parameters

- config – Pointer to the configuration.

uint32_t CLOCK_GetLpavSysClkFreq(*cgc_sys_clk_t* type)

Gets the CGC NIC LPAV system clock frequency in LPAV.

This function gets the CGC NIC LPAV system clock frequency. These clocks are used for AXI, AHB, Bus domains.

Parameters

- type – Which type of clock to get.

Returns

Clock frequency.

static inline void CLOCK_SetLpavSysClkConfig(const *cgc_lpav_sys_clk_config_t* *config)

Sets the system clock configuration for NIC LPAV domain.

This function sets the system clock configuration for NIC LPAV domain.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetLpavSysClkConfig(*cgc_lpav_sys_clk_config_t* *config)

Gets the system clock configuration for NIC LPAV domain.

This function gets the system configuration for NIC LPAV domain.

Parameters

- config – Pointer to the configuration.

uint32_t CLOCK_GetDdrClkFreq(void)

Gets the CGC DDR clock frequency in LPAV.

This function gets the CGC DDR clock frequency.

Returns

Clock frequency.

```
static inline void CLOCK_SetLpavClkOutConfig(clock_lpav_clkout_src_t setting, uint8_t div, bool enable)
```

Sets the clock out configuration in LPAV.

This function sets the clock out configuration.

Parameters

- setting – The selection to set.
- div – The divider to set (div > 0).
- enable – Enable clock out.

```
status_t CLOCK_InitSysOsc(const cgc_sosc_config_t *config)
```

Initializes the CGC system OSC.

This function enables the CGC system OSC clock according to the configuration.

Note: This function can't detect whether the system OSC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – System OSC is initialized.
- kStatus_CGC_Busy – System OSC has been enabled and is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

```
status_t CLOCK_DeinitSysOsc(void)
```

De-initializes the CGC system OSC.

This function disables the CGC system OSC clock.

Note: This function can't detect whether the system OSC is used by an IP.

Return values

- kStatus_Success – System OSC is deinitialized.
- kStatus_CGC_Busy – System OSC is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

```
void CLOCK_SetRtdSysOscAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)
```

Set the asynchronous clock divider in RTD.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- asyncClk – Which asynchronous clock to configure.
- divider – The divider value to set. Disabled when divider == 0.

`void CLOCK_SetAdSysOscAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`
Set the asynchronous clock divider in AD.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`void CLOCK_SetLpavSysOscAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`
Set the asynchronous clock divider in LPAV.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`uint32_t CLOCK_GetSysOscFreq(void)`
Gets the CGC system OSC clock frequency (SYSOSC).

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t CLOCK_GetRtdSysOscAsyncFreq(cgc_async_clk_t type)`
Gets the CGC asynchronous clock frequency from the system OSC in RTD.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t CLOCK_GetAdSysOscAsyncFreq(cgc_async_clk_t type)`
Gets the CGC asynchronous clock frequency from the system OSC in AD.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t CLOCK_GetLpavSysOscAsyncFreq(cgc_async_clk_t type)`
Gets the CGC asynchronous clock frequency from the system OSC in LPAV.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsSysOscErr(void)

Checks whether the system OSC clock error occurs.

Returns

True if the error occurs, false if not.

static inline void CLOCK_ClearSysOscErr(void)

Clears the system OSC clock error.

static inline void CLOCK_SetSysOscMonitorMode(*cgc_sosc_monitor_mode_t* mode)

Sets the system OSC monitor mode.

This function sets the system OSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

static inline bool CLOCK_IsSysOscSelected(void)

Checks whether the system OSC clock is used as clock source.

Returns

True if system OSC is used as clock source, false if not.

static inline bool CLOCK_IsSysOscValid(void)

Checks whether the system OSC clock is valid.

Returns

True if clock is valid, false if not.

status_t CLOCK_InitFro(const *cgc_fro_config_t* *config)

Initializes the CGC FRO clock.

This function initializes the CGC FRO clock according to the configuration.

Note: This function can't detect whether the FRO has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – FRO is initialized.
- kStatus_CGC_Busy – FRO has been enabled and is used by system clock.
- kStatus_ReadOnly – FRO control register is locked.

status_t CLOCK_DeinitFro(void)

De-initializes the CGC FRO.

This function deinitializes the CGC FRO.

Note: This function can't detect whether the FRO is used by an IP.

Return values

- kStatus_Success – FRO is deinitialized.
- kStatus_CGC_Busy – FRO is used by system clock.
- kStatus_ReadOnly – FRO control register is locked.

`void CLOCK_SetRtdFroAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`

Set the asynchronous clock divider in RTD.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`void CLOCK_SetAdFroAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`

Set the asynchronous clock divider in AD.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`void CLOCK_SetLpavFroAsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`

Set the asynchronous clock divider in LPAV.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`void CLOCK_EnableFroTuning(bool enable)`

Enable/Disable FRO tuning. On enable, the function will wait until FRO is close to the target frequency.

`uint32_t CLOCK_GetFroFreq(void)`

Gets the CGC FRO clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t CLOCK_GetRtdFroAsyncFreq(cgc_async_clk_t type)`

Gets the CGC asynchronous clock frequency from the FRO in RTD.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t CLOCK_GetAdFroAsyncFreq(cgc_async_clk_t type)`

Gets the CGC asynchronous clock frequency from the FRO in AD.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t` `CLOCK__GetLpavFroAsyncFreq(cgc_async_clk_t type)`

Gets the CGC asynchronous clock frequency from the FRO in LPAV.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`static inline bool` `CLOCK__IsFroSelected(void)`

Checks whether the FRO clock is used as clock source.

Returns

True if FRO is used as clock source, false if not.

`static inline bool` `CLOCK__IsFroValid(void)`

Checks whether the FRO clock is valid.

Returns

True if clock is valid, false if not.

`status_t` `CLOCK__InitLposc(const cgc_lposc_config_t *config)`

Initializes the CGC LPOSC clock.

This function initializes the CGC LPOSC clock according to the configuration.

Note: This function can't detect whether the LPOSC has been enabled and used by an IP.

Parameters

- `config` – Pointer to the configuration structure.

Return values

- `kStatus__Success` – LPOSC is initialized.
- `kStatus__ReadOnly` – FRO control register is locked.

`status_t` `CLOCK__DeinitLposc(void)`

De-initializes the CGC LPOSC.

This function deinitializes the CGC LPOSC.

Note: This function can't detect whether the LPOSC is used by an IP.

Return values

- `kStatus__Success` – LPOSC is deinitialized.
- `kStatus__ReadOnly` – LPOSC control register is locked.

`static inline bool` `CLOCK__IsLpOscValid(void)`

Checks whether the LPOSC clock is valid.

Returns

True if clock is valid, false if not.

uint32_t CLOCK_GetLpOscFreq(void)

Gets the CGC LPOSC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetRtcOscFreq(void)

Gets the CGC RTC OSC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsRtcOscErr(void)

Checks whether the RTC OSC clock error occurs.

Returns

True if error occurs, false if not.

static inline void CLOCK_ClearRtcOscErr(void)

Clears the RTC OSC clock error.

void CLOCK_SetRtcOscMonitorMode(*cgc_rosc_monitor_mode_t* mode)

Sets the RTC OSC monitor mode.

This function sets the RTC OSC monitor mode. The mode can be disabled. It can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

static inline bool CLOCK_IsRtcOscSelected(void)

Checks whether the RTCOSC clock is used as clock source.

Returns

True if RTCOSC is used as clock source, false if not.

static inline bool CLOCK_IsRtcOscValid(void)

Checks whether the RTC OSC clock is valid.

Returns

True if the clock is valid, false if not.

status_t CLOCK_InitPll0(const *cgc_pll0_config_t* *config)

Initializes the CGC PLL0.

This function enables the CGC PLL0 clock according to the configuration. The PLL0 can use the OSC or FRO as the clock source. Ensure that the source clock is valid before calling this function.

Example code for initializing PLL0 clock output:

```
const cgc_pll0_config_t g_cgcPll0Config = {.enableMode = kCGC_PllEnable,  
                                           .div1 = 1U,  
                                           .pfd1Div = 2U,  
                                           .pfd2Div = 0U,  
                                           .src = kCGC_PllSrcSysOsc,  
                                           .mult = kCGC_Pll0Mult20};  
CLOCK_InitPll0(&g_cgcPll0Config);
```

Note: This function can't detect whether the PLL0 has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus__Success – PLL0 is initialized.
- kStatus__CGC__Busy – PLL0 has been enabled and is used by the system clock.
- kStatus__ReadOnly – PLL0 control register is locked.

status_t CLOCK_DeinitPll0(void)

De-initializes the CGC PLL0.

This function disables the CGC PLL0.

Note: This function can't detect whether the PLL0 is used by an IP.

Return values

- kStatus__Success – PLL0 is deinitialized.
- kStatus__CGC__Busy – PLL0 is used by the system clock.
- kStatus__ReadOnly – PLL0 control register is locked.

void CLOCK_SetPll0AsyncClkDiv(*cgc_async_clk_t* asyncClk, uint8_t divider)

Set the asynchronous clock divider.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- asyncClk – Which asynchronous clock to configure.
- divider – The divider value to set. Disabled when divider == 0.

uint32_t CLOCK_GetPll0Freq(void)

Gets the CGC PLL0 clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetPll0AsyncFreq(*cgc_async_clk_t* type)

Gets the CGC asynchronous clock frequency from the PLL0.

Parameters

- type – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetPll0PfdFreq(*cgc_pll_pfd_clkout_t* pfdClkout)

Gets the CGC PLL0 PFD clock frequency.

Parameters

- pfdClkout – The selected PFD clock out. See “cgc_pll_pfd_clkout_t”.

Returns

Clock frequency; If the clock is invalid, returns 0.

`void CLOCK_EnablePll0PfdClkout(cgc_pll_pfd_clkout_t pfdClkout, uint8_t fracValue)`

Enables the CGC PLL0 Fractional Divide (PFD) clock out with configurations.

PLL Frequency = Fref * MULT PFD Clock Frequency = PLL output frequency * 18/frac value

Example code for configuring PLL0 PFD0 clock output:

```
const cgc_pll0_config_t g_cgcPll0Config = {.enableMode = kCGC_PllEnable,
                                           .div1 = 1U,
                                           .pfd1Div = 2U,
                                           .pfd2Div = 0U,
                                           .src = kCGC_PllSrcSysOsc,
                                           .mult = kCGC_Pll0Mult20};
CLOCK_InitPll0(&g_cgcPll0Config);
CLOCK_EnablePll0PfdClkout(kCGC_PllPfd0Clk, 15U);
```

Parameters

- pfdClkout – PLL0 PFD clock out select.
- fracValue – Fractional Divider value. Recommended to be kept between 12-35 for all PFDs.

`static inline void CLOCK_DisablePll0PfdClkout(cgc_pll_pfd_clkout_t pfdClkout)`

Disables the CGC PLL0 Fractional Divide (PFD) clock out.

`static inline void CLOCK_SetPll0LockTime(uint16_t lockTime)`

Sets the CGC PLL0 lock time.

Parameters

- lockTime – Reference clocks to count before PLL0 is considered locked and valid.

`static inline bool CLOCK_IsPll0Selected(void)`

Checks whether the PLL0 clock is used as clock source.

Returns

True if PLL0 is used as clock source, false if not.

`static inline bool CLOCK_IsPll0Valid(void)`

Checks whether the PLL0 clock is valid.

Returns

True if the clock is valid, false if not.

`status_t CLOCK_InitPll1(const cgc_pll1_config_t *config)`

Initializes the CGC PLL1.

This function enables the CGC PLL1 clock according to the configuration. The PLL1 can use the system OSC or FRO as the clock source. Ensure that the source clock is valid before calling this function.

Example code for initializing PLL1 clock output:

```
const cgc_pll1_config_t g_cgcPll1Config = {.enableMode = kCGC_PllEnable,
                                           .div1 = 0U,
                                           .pfd1Div = 0U,
                                           .pfd2Div = 0U,
                                           .src = kCGC_PllSrcFro24M,
                                           .mult = kCGC_Pll1Mult22,
                                           .num = 578,
                                           .denom = 1000};
CLOCK_InitPll1(&g_cgcPll1Config);
```

Note: This function can't detect whether the PLL1 has been enabled and is used by an IP.

Parameters

- `config` – Pointer to the configuration structure.

Return values

- `kStatus_Success` – PLL1 is initialized.
- `kStatus_CGC_Busy` – PLL1 has been enabled and is used by the system clock.
- `kStatus_ReadOnly` – PLL1 control register is locked.

`status_t` `CLOCK_DeinitPll1(void)`

De-initializes the CGC PLL1.

This function disables the CGC PLL1.

Note: This function can't detect whether the PLL1 is used by an IP.

Return values

- `kStatus_Success` – PLL1 is deinitialized.
- `kStatus_CGC_Busy` – PLL1 is used by the system clock.
- `kStatus_ReadOnly` – PLL1 control register is locked.

`void` `CLOCK_SetPll1AsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`

Set the asynchronous clock divider.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`uint32_t` `CLOCK_GetPll1Freq(void)`

Gets the CGC PLL1 clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t` `CLOCK_GetPll1AsyncFreq(cgc_async_clk_t type)`

Gets the CGC asynchronous clock frequency from the PLL1.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t` `CLOCK_GetPll1PfdFreq(cgc_pll_pfd_clkout_t pfdClkout)`

Gets the CGC PLL1 PFD clock frequency.

Parameters

- pfdClkout – The selected PFD clocks out. See “cgc_pll_pfd_clkout_t”.

Returns

Clock frequency; If the clock is invalid, returns 0.

void CLOCK_EnablePll1PfdClkout(*cgc_pll_pfd_clkout_t* pfdClkout, uint8_t fracValue)

Enables the CGC PLL1 Fractional Divide (PFD) clock out with configurations.

PLL1 Frequency = Fref * (MULT + NUM/DENOM) PFD Clock Frequency = PLL output frequency * 18/frac value

Example code for configuring PLL1 as PLL1 PFD clock output:

```
const cgc_pll1_config_t g_cgcPll1Config = {.enableMode = kCGC_PllEnable,
                                           .div1 = 0U,
                                           .pfd1Div = 0U,
                                           .pfd2Div = 0U,
                                           .src = kCGC_PllSrcFro24M,
                                           .mult = kCGC_Pll1Mult22,
                                           .num = 578,
                                           .denom = 1000};
CLOCK_InitPll1(&g_cgcPll1Config);
CLOCK_EnablePll1PfdClkout(kCGC_PllPfd0Clk, 15U);
```

Parameters

- pfdClkout – PLL1 PFD clock out select.
- fracValue – Fractional Divider value. Recommended to be kept between 12-35 for all PFDs.

static inline void CLOCK_DisablePll1PfdClkout(*cgc_pll_pfd_clkout_t* pfdClkout)

Disables the CGC PLL1 Fractional Divide (PFD) clock out.

static inline void CLOCK_EnablePll1SpectrumModulation(uint16_t step, uint16_t stop)

Enables the CGC PLL1 spread spectrum modulation feature with configurations.

This function sets the CGC PLL1 spread spectrum modulation configurations. STOP and STEP together control the modulation depth (maximum frequency change) and modulation frequency.

Modulation Depth = (STOP/MFD)*Fref where MFD is the DENOM field value in DENOM register. Modulation Frequency = (STEP/(2*STOP))*Fref.

Parameters

- step – PLL1 Spread Spectrum STEP.
- stop – PLL1 Spread Spectrum STOP.

static inline void CLOCK_DisablePll1SpectrumModulation(void)

Disables the CGC PLL1 spread spectrum modulation.

static inline void CLOCK_SetPll1LockTime(uint16_t lockTime)

Sets the CGC PLL1 lock time.

Parameters

- lockTime – Reference clocks to count before PLL1 is considered locked and valid.

static inline bool CLOCK_IsPll1Selected(void)

Checks whether the PLL1 clock is used as clock source.

Returns

True if PLL1 is used as clock source, false if not.

static inline bool CLOCK_IsPll1Valid(void)

Checks whether the PLL1 clock is valid.

Returns

True if the clock is valid, false if not.

uint32_t CLOCK_GetPll3Freq(void)

Gets the CGC PLL3 clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetPll3AsyncFreq(*cgc_async_clk_t* type)

Gets the CGC asynchronous clock frequency from the PLL3.

Parameters

- type – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetPll3PfdFreq(*cgc_pll_pfd_clkout_t* pfdClkout)

Gets the CGC PLL3 PFD clock frequency.

Parameters

- pfdClkout – The selected PFD clock out. See “cgc_pll_pfd_clkout_t”.

Returns

Clock frequency; If the clock is invalid, returns 0.

status_t CLOCK_InitPll4(const *cgc_pll4_config_t* *config)

Initializes the CGC PLL4.

This function enables the CGC PLL4 clock according to the configuration. The PLL4 can use the OSC or FRO as the clock source. Ensure that the source clock is valid before calling this function.

Example code for initializing PLL4 clock output:

```
const cgc_pll4_config_t g_cgcPll4Config = {enableMode = kCGC_PllEnable,
    .div1 = 0U,
    .pfd0Div1 = 0U,
    .pfd0Div2 = 0U,
    .pfd1Div1 = 0U,
    .pfd1Div2 = 0U,
    .pfd2Div1 = 0U,
    .pfd2Div2 = 0U,
    .pfd3Div1 = 0U,
    .pfd3Div2 = 0U,
    .src = kCGC_PllSrcFro24M,
    .mult = kCGC_Pll4Mult22,
    .num = 578,
    .denom = 1000};
CLOCK_InitPll4(&g_cgcPll4Config);
```

Note: This function can't detect whether the PLL4 has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- `kStatus_Success` – PLL4 is initialized.
- `kStatus_CGC_Busy` – PLL4 has been enabled and is used by the system clock.
- `kStatus_ReadOnly` – PLL4 control register is locked.

`status_t` `CLOCK_DeinitPll4(void)`

De-initializes the CGC PLL4.

This function disables the CGC PLL4.

Note: This function can't detect whether the PLL4 is used by an IP.

Return values

- `kStatus_Success` – PLL4 is deinitialized.
- `kStatus_CGC_Busy` – PLL4 is used by the system clock.
- `kStatus_ReadOnly` – PLL4 control register is locked.

`void` `CLOCK_SetPll4AsyncClkDiv(cgc_async_clk_t asyncClk, uint8_t divider)`

Set the asynchronous clock divider.

Note: There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

Parameters

- `asyncClk` – Which asynchronous clock to configure.
- `divider` – The divider value to set. Disabled when `divider == 0`.

`uint32_t` `CLOCK_GetPll4Freq(void)`

Gets the CGC PLL4 clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t` `CLOCK_GetPll4AsyncFreq(cgc_async_clk_t type)`

Gets the CGC asynchronous clock frequency from the PLL4.

Parameters

- `type` – The asynchronous clock type.

Returns

Clock frequency; If the clock is invalid, returns 0.

`uint32_t` `CLOCK_GetPll4PfdFreq(cgc_pll_pfd_clkout_t pfdClkout)`

Gets the CGC PLL4 PFD clock frequency.

Parameters

- `pfdClkout` – The selected PFD clock out. See “`cgc_pll_pfd_clkout_t`”.

Returns

Clock frequency; If the clock is invalid, returns 0.

`void` `CLOCK_EnablePll4PfdClkout(cgc_pll_pfd_clkout_t pfdClkout, uint8_t fracValue)`

Enables the CGC PLL4 Fractional Divide (PFD) clock out with configurations.

PLL Frequency = Fref * MULT PFD Clock Frequency = PLL output frequency * 18/frac value

Example code for configuring PLL4 PFD0 clock output:

```
const cgc_pll4_config_t g_cgcPll4Config = {.enableMode = kCGC_PllEnable,  
                                           .div1 = 0U,  
                                           .pfd0Div1 = 0U,  
                                           .pfd0Div2 = 0U,  
                                           .pfd1Div1 = 0U,  
                                           .pfd1Div2 = 0U,  
                                           .pfd2Div1 = 0U,  
                                           .pfd2Div2 = 0U,  
                                           .pfd3Div1 = 0U,  
                                           .pfd3Div2 = 0U,  
                                           .src = kCGC_PllSrcFro24M,  
                                           .mult = kCGC_Pll4Mult22,  
                                           .num = 578,  
                                           .denom = 1000};  
CLOCK_InitPll4(&g_cgcPll4Config);  
CLOCK_EnablePll4PfdClkout(kCGC_PllPfd0Clk, 15U);
```

Parameters

- pfdClkout – PLL4 PFD clock out select.
- fracValue – Fractional Divider value. Recommended to be kept between 12-35 for all PFDs.

static inline void CLOCK_DisablePll4PfdClkout(*cgc_pll_pfd_clkout_t* pfdClkout)

Disables the CGC PLL4 Fractional Divide (PFD) clock out.

static inline void CLOCK_EnablePll4SpectrumModulation(uint16_t step, uint16_t stop)

Enables the CGC PLL4 spread spectrum modulation feature with configurations.

This function sets the CGC PLL4 spread spectrum modulation configurations. STOP and STEP together control the modulation depth (maximum frequency change) and modulation frequency.

Modulation Depth = (STOP/MFD)*Fref where MFD is the DENOM field value in DENOM register. Modulation Frequency = (STEP/(2*STOP))*Fref.

Parameters

- step – PLL4 Spread Spectrum STEP.
- stop – PLL4 Spread Spectrum STOP.

static inline void CLOCK_SetPll4LockTime(uint16_t lockTime)

Sets the CGC PLL4 lock time.

Parameters

- lockTime – Reference clocks to count before PLL4 is considered locked and valid.

static inline bool CLOCK_IsPll4Selected(void)

Checks whether the PLL4 clock is used as clock source.

Returns

True if PLL4 is used as clock source, false if not.

static inline bool CLOCK_IsPll4Valid(void)

Checks whether the PLL4 clock is valid.

Returns

True if the clock is valid, false if not.

static inline void CLOCK_SetXtal0Freq(uint32_t freq)

Sets the XTAL0 frequency based on board settings.

Parameters

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

static inline void CLOCK_SetXtal32Freq(uint32_t freq)

Sets the XTAL32 frequency based on board settings.

Parameters

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

static inline void CLOCK_SetLvdsFreq(uint32_t freq)

Sets the LVDS pad frequency based on board settings.

Parameters

- freq – The LVDS pad input clock frequency in Hz.

static inline void CLOCK_SetMclkFreq(uint32_t index, uint32_t freq)

Sets the MCLK pad frequency based on Audio settings.

Parameters

- index – The MCLK index.
- freq – The MCLK pad input clock frequency in Hz.

static inline void CLOCK_SetRxBclkFreq(uint32_t instance, uint32_t freq)

Sets the RX_BCLK pad frequency based on Audio settings.

Parameters

- instance – The SAI instance to contribute to this RX_BCLK pad.
- freq – The RX_BCLK pad input clock frequency in Hz.

static inline void CLOCK_SetTxBclkFreq(uint32_t instance, uint32_t freq)

Sets the TX_BCLK pad frequency based on Audio settings.

Parameters

- instance – The SAI instance to contribute to this TX_BCLK pad.
- freq – The TX_BCLK pad input clock frequency in Hz.

static inline void CLOCK_SetSpdifRxFreq(uint32_t freq)

Sets the SPDIF_RX frequency based on Audio settings.

Parameters

- freq – The SPDIF_RX input clock frequency in Hz.

uint32_t CLOCK_GetWdogClkFreq(uint32_t instance)

Gets the WDOG clock frequency in RTD and LPAV.

Parameters

- instance – The WDOG instance (0-2,5).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetFlexspiClkFreq(uint32_t instance)

Gets the FlexSPI clock frequency in RTD.

Parameters

- instance – The FlexSPI instance (0-1).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetLpitClkFreq(void)

Gets the LPIT clock frequency in RTD.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetFlexioClkFreq(void)

Gets the FlexIO clock frequency in RTD.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetI3cClkFreq(uint32_t instance)

Gets the I3C clock frequency in RTD and LPAV.

Parameters

- instance – The I3C instance (0-1).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetLpspiClkFreq(uint32_t instance)

Gets the LPSPI clock frequency in RTD.

Parameters

- instance – The LPSPI instance (0-3).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetAdcClkFreq(uint32_t instance)

Gets the ADC clock frequency.

Parameters

- instance – The ADC instance (0-1).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetDacClkFreq(uint32_t instance)

Gets the DAC clock frequency.

Parameters

- instance – The DAC instance (0-1).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetTpiuClkFreq(void)

Gets the TPIU clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetSwoClkFreq(void)

Gets the SWO clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetTpmClkFreq(uint32_t instance)

Gets the TPM clock frequency.

Parameters

- instance – The TPM instance (0-8).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetLpi2cClkFreq(uint32_t instance)

Gets the LPI2C clock frequency in RTD.

Parameters

- instance – The LPI2C instance (0-3).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetLpuartClkFreq(uint32_t instance)

Gets the LPUART clock frequency in RTD.

Parameters

- instance – The LPUART instance (0-3).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetFlexcanClkFreq(void)

Gets the FlexCAN clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetCsiClkFreq(void)

Gets the CSI clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetDsiClkFreq(void)

Gets the DSI clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetEpdcClkFreq(void)

Gets the EPDC clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetGpu2dClkFreq(void)

Gets the GPU2D clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetGpu3dClkFreq(void)

Gets the GPU3D clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetDcnanoClkFreq(void)

Gets the DC Nano clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetCsiUiClkFreq(void)

Gets the CSI clk_ui clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetCsiEscClkFreq(void)

Gets the CSI clk_esc clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetRtdAudClkFreq(void)

Gets the audio clock frequency in RTD.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetAdAudClkFreq(void)

Gets the audio clock frequency in AD.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetLpavAudClkFreq(void)

Gets the audio clock frequency in LPAV.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetSaiFreq(uint32_t instance)

Gets the SAI clock frequency.

Parameters

- instance – The SAI instance (0-7).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetSpdifFreq(void)

Gets the SPDIF clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetMqsFreq(uint32_t instance)

Gets the MQS clock frequency.

Parameters

- instance – The MQS instance (0-1).

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetMicfilFreq(void)

Gets the EMICFIL clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetMrtFreq(void)

Gets the MRT clock frequency.

Returns

Clock frequency; If the clock is invalid, returns 0.

uint32_t CLOCK_GetRtdSysClkFreq(uint32_t config, *cgc_sys_clk_t* type)

Gets the CGC system clock frequency in RTD.

This function gets the CGC system clock frequency. These clocks are used for core, platform, bus and slow clock domains.

Parameters

- config – Config value from CGC register.
- type – Which type of clock to get.

Returns

Clock frequency.

uint32_t divSlow

Slow clock divider, selected division is the value of the field + 1

uint32_t ____pad0____

Reserved.

uint32_t divBus

Bus clock divider, selected division is the value of the field + 1.

uint32_t ____pad1____

Reserved.

uint32_t divCore

Core/Platform clock divider, selected division is the value of the field + 1.

uint32_t switchFin

1: Clock is running. 0: Clock is not running.

uint32_t src

System clock source, see *cgc_rtd_sys_clk_src_t*.

uint32_t locked

Clock register locked.

uint32_t ____pad0____

Reserved.

uint32_t divPlat

Platform clock divider, selected division is the value of the field + 1.

uint32_t ____pad1____

Reserved.

uint32_t divCore

Core clock divider, selected division is the value of the field + 1.

uint32_t switchFin

1: Clock is running. 0: Clock is not running.

uint32_t src

System clock source, see *cgc_hifi_sys_clk_src_t*.

uint32_t locked

Clock register locked.

uint32_t __pad0__

Reserved.

uint32_t divBus

Platform clock divider, selected division is the value of the field + 1.

uint32_t __pad1__

Reserved.

uint32_t divAhb

Platform clock divider, selected division is the value of the field + 1.

uint32_t __pad2__

Reserved.

uint32_t divAxi

Core clock divider, selected division is the value of the field + 1.

uint32_t switchFin

1: Clock is running. 0: Clock is not running.

uint32_t src

System clock source, see `cgc_lpav_sys_clk_src_t`.

uint32_t __pad3__

Reserved.

uint32_t locked

Clock register locked.

uint32_t __pad0__

Reserved.

uint32_t divDdr

DDR clock divider, selected division is the value of the field + 1.

uint32_t switchFin

1: Clock is running. 0: Clock is not running.

uint32_t src

System clock source, see `cgc_lpav_sys_clk_src_t`.

uint32_t locked

Clock register locked.

uint32_t freq

System OSC frequency.

cgc_sosc_monitor_mode_t monitorMode

Clock monitor mode selected.

uint8_t enableMode

Enable mode, OR'ed value of `_cgc_sosc_enable_mode`.

cgc_sosc_mode_t workMode

OSC work mode.

uint32_t enableMode

Enable mode, OR'ed value of `_cgc_fro_enable_mode`.

uint32_t enableMode

Enable mode, OR'ed value of `_cgc_lposc_enable_mode`.

`uint8_t enableMode`

Enable mode, OR'ed value of `_cgc_pll_enable_mode`

`uint8_t div1`

PLLDIV_VCO divider value. Disabled when `div1 == 0`.

`uint8_t pfd1Div`

PLLDIV_PFD_0 DIV1 divider value. Disabled when `pfd1Div == 0`.

`uint8_t pfd2Div`

PLLDIV_PFD_0 DIV2 divider value. Disabled when `pfd2Div == 0`.

`cgc_pll_src_t src`

Clock source.

`cgc_pll0_mult_t mult`

PLL multiplier.

`cgc_rosc_monitor_mode_t monitorMode`

Clock monitor mode selected.

`uint8_t enableMode`

Enable mode, OR'ed value of `_cgc_pll_enable_mode`

`uint8_t div1`

PLLDIV_VCO divider value. Disabled when `div1 == 0`.

`uint8_t pfd1Div`

PLLDIV_PFD_0 DIV1 divider value. Disabled when `pfd1Div == 0`.

`uint8_t pfd2Div`

PLLDIV_PFD_0 DIV2 divider value. Disabled when `pfd2Div == 0`.

`cgc_pll_src_t src`

Clock source.

`cgc_pll1_mult_t mult`

PLL1 multiplier.

`uint32_t num`

30-bit numerator of the PLL1 Fractional-Loop divider.

`uint32_t denom`

30-bit denominator of the PLL1 Fractional-Loop divider.

`uint8_t enableMode`

Enable mode, OR'ed value of `_cgc_pll_enable_mode`

`uint8_t div1`

PLLDIV_VCO divider value. Disabled when `div1 == 0`.

`uint8_t pfd0Div1`

PLLDIV_PFD_0 DIV1 divider value. Disabled when `pfd0Div1 == 0`.

`uint8_t pfd0Div2`

PLLDIV_PFD_0 DIV2 divider value. Disabled when `pfd0Div2 == 0`.

`uint8_t pfd1Div1`

PLLDIV_PFD_0 DIV3 divider value. Disabled when `pfd1Div1 == 0`.

`uint8_t pfd1Div2`

PLLDIV_PFD_0 DIV4 divider value. Disabled when `pfd1Div2 == 0`.

uint8_t pfd2Div1

PLLDIV_PFD_1 DIV1 divider value. Disabled when pfd2Div1 == 0.

uint8_t pfd2Div2

PLLDIV_PFD_1 DIV2 divider value. Disabled when pfd2Div2 == 0.

uint8_t pfd3Div1

PLLDIV_PFD_2 DIV3 divider value. Disabled when pfd3Div1 == 0.

uint8_t pfd3Div2

PLLDIV_PFD_2 DIV4 divider value. Disabled when pfd3Div2 == 0.

cgc_pll_src_t src

Clock source.

cgc_pll4_mult_t mult

PLL4 multiplier.

uint32_t num

30-bit numerator of the PLL4 Fractional-Loop divider.

uint32_t denom

30-bit denominator of the PLL4 Fractional-Loop divider.

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 _cgc_rtd_sys_clk_config

#include <fsl_clock.h> CGC system clock configuration for RTD.

struct _cgc_hifi_sys_clk_config

#include <fsl_clock.h> CGC system clock configuration for HIFI4 DSP.

struct _cgc_lpav_sys_clk_config

#include <fsl_clock.h> CGC system clock configuration for LPAV.

struct _cgc_ddr_sys_clk_config

#include <fsl_clock.h> CGC system clock configuration for DDR in LPAV.

struct _cgc_sosc_config

#include <fsl_clock.h> CGC system OSC configuration.

struct _cgc_fro_config

#include <fsl_clock.h> CGC FRO clock configuration.

struct _cgc_lposc_config

#include <fsl_clock.h> CGC LPOSC clock configuration.

struct _cgc_pll0_config

#include <fsl_clock.h> CGC PLL0 configuration.

struct _cgc_rosc_config

#include <fsl_clock.h> CGC RTC OSC configuration.

```
struct _cgc_pll1_config
    #include <fsl_clock.h> CGC PLL1 configuration.
struct _cgc_pll4_config
    #include <fsl_clock.h> CGC PLL4 configuration.
```

2.8 CMC: Core Mode Controller Driver

```
void CMC_SetClockMode(CMC_Type *base, cmc_clock_mode_t mode)
```

Sets clock mode.

This function config the amount of clock gating when the core asserts Sleeping due to WFI, WFE or SLEEPONEXIT.

Parameters

- base – CMC peripheral base address.
- mode – System clock mode.

```
static inline void CMC_LockClockModeSetting(CMC_Type *base)
```

Locks the clock mode setting.

After invoking this function, any clock mode setting will be blocked.

Parameters

- base – CMC peripheral base address.

```
static inline cmc_core_clock_gate_status_t CMC_GetCoreClockGatedStatus(CMC_Type *base)
```

Gets the core clock gated status.

This function get the status to indicate whether the core clock is gated. The core clock gated status can be cleared by software.

Parameters

- base – CMC peripheral base address.

Returns

The status to indicate whether the core clock is gated.

```
static inline void CMC_ClearCoreClockGatedStatus(CMC_Type *base)
```

Clears the core clock gated status.

This function clear clock status flag by software.

Parameters

- base – CMC peripheral base address.

```
static inline uint8_t CMC_GetWakeupSource(CMC_Type *base)
```

Gets the Wakeup Source.

This function gets the Wakeup sources from the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns

The Wakeup sources from the previous low power mode entry. See `_cmc_wakeup_sources` for details.

```
static inline cmc_clock_mode_t CMC_GetClockMode(CMC_Type *base)
```

Gets the Clock mode.

This function gets the clock mode of the previous low power mode entry.

Parameters

- `base` – CMC peripheral base address.

Returns

The Low Power status.

```
static inline uint32_t CMC_GetSystemResetStatus(CMC_Type *base)
```

Gets the System reset status.

This function returns the system reset status. Those status updates on every MAIN Warm Reset to indicate the type/source of the most recent reset.

Parameters

- `base` – CMC peripheral base address.

Returns

The most recent system reset status. See `_cmc_system_reset_sources` for details.

```
static inline uint32_t CMC_GetStickySystemResetStatus(CMC_Type *base)
```

Gets the sticky system reset status since the last WAKE Cold Reset.

This function gets all source of system reset that have generated a system reset since the last WAKE Cold Reset, and that have not been cleared by software.

Parameters

- `base` – CMC peripheral base address.

Returns

System reset status that have not been cleared by software. See `_cmc_system_reset_sources` for details.

```
static inline void CMC_ClearStickySystemResetStatus(CMC_Type *base, uint32_t mask)
```

Clears the sticky system reset status flags.

Parameters

- `base` – CMC peripheral base address.
- `mask` – Bitmap of the sticky system reset status to be cleared.

```
static inline uint8_t CMC_GetResetCount(CMC_Type *base)
```

Gets the number of reset sequences completed since the last WAKE Cold Reset.

Parameters

- `base` – CMC peripheral base address.

Returns

The number of reset sequences.

```
void CMC_SetPowerModeProtection(CMC_Type *base, uint32_t allowedModes)
```

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes. This should be done before set the lowPower mode for each power doamin.

The allowed lowpower modes are passed as bit map. For example, to allow Sleep and DeepSleep, use `CMC_SetPowerModeProtection(CMC_base, kCMC_AllowSleepMode | kCMC_AllowDeepSleepMode)`. To allow all low power modes, use `CMC_SetPowerModeProtection(CMC_base, kCMC_AllowAllLowPowerModes)`.

Parameters

- base – CMC peripheral base address.
- allowedModes – Bitmaps of the allowed power modes. See `_cmc_power_mode_protection` for details.

static inline void CMC_LockPowerModeProtectionSetting(CMC_Type *base)

Locks the power mode protection.

This function locks the power mode protection. After invoking this function, any power mode protection setting will be ignored.

Parameters

- base – CMC peripheral base address.

static inline void CMC_SetGlobalPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Config the same lowPower mode for all power domain.

This function configures the same low power mode for MAIN power domain and WAKE power domain.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

static inline void CMC_SetMAINPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Configures entry into low power mode for the MAIN Power domain.

This function configures the low power mode for the MAIN power domain, when the core executes WFI/WFE instruction. The available lowPower modes are defined in the `cmc_low_power_mode_t`.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

static inline *cmc_low_power_mode_t* CMC_GetMAINPowerMode(CMC_Type *base)

Gets the power mode of the MAIN Power domain.

Parameters

- base – CMC peripheral base address.

Returns

The power mode of MAIN Power domain. See `cmc_low_power_mode_t` for details.

static inline void CMC_SetWAKEPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Configure entry into low power mode for the WAKE Power domain.

This function configures the low power mode for the WAKE power domain, when the core executes WFI/WFE instruction. The available lowPower mode are defined in the `cmc_low_power_mode_t`.

Note: The lowPower Mode for the WAKE domain must not be configured to a lower power mode than any other power domain.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

static inline `cmc_low_power_mode_t` CMC_GetWAKEPowerMode(CMC_Type *base)

Gets the power mode of the WAKE Power domain.

Parameters

- base – CMC peripheral base address.

Returns

The power mode of WAKE Power domain. See `cmc_low_power_mode_t` for details.

void CMC_ConfigResetPin(CMC_Type *base, const `cmc_reset_pin_config_t` *config)

Configure reset pin.

This function configures reset pin. When enabled, the low power filter is enabled in both Active and Low power modes, the reset filter is only enabled in Active mode. When both filters are enabled, they operate in series.

Parameters

- base – CMC peripheral base address.
- config – Pointer to the reset pin config structure.

static inline void CMC_EnableSystemResetInterrupt(CMC_Type *base, uint32_t mask)

Enable system reset interrupts.

This function enables the system reset interrupts. The assertion of non-fatal warm reset can be delayed for 258 cycles of the 32K_CLK clock while an enabled interrupt is generated. Then Software can perform a graceful shutdown or abort the non-fatal warm reset provided the pending reset source is cleared by resetting the reset source and then clearing the pending flag.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See `_cmc_system_reset_interrupt_enable` for details.

static inline void CMC_DisableSystemResetInterrupt(CMC_Type *base, uint32_t mask)

Disable system reset interrupts.

This function disables the system reset interrupts.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See `_cmc_system_reset_interrupt_enable` for details.

static inline uint32_t CMC_GetSystemResetInterruptFlags(CMC_Type *base)

Gets System Reset interrupt flags.

This function returns the System reset interrupt flags.

Parameters

- base – CMC peripheral base address.

Returns

System reset interrupt flags. See `_cmc_system_reset_interrupt_flag` for details.

```
static inline void CMC_ClearSystemResetInterruptFlags(CMC_Type *base, uint32_t mask)
```

Clears System Reset interrupt flags.

This function clears system reset interrupt flags. The pending reset source can be cleared by resetting the source of the reset and then clearing the pending flags.

Parameters

- base – CMC peripheral base address.
- mask – System Reset interrupt flags. See `_cmc_system_reset_interrupt_flag` for details.

```
static inline void CMC_EnableNonMaskablePinInterrupt(CMC_Type *base, bool enable)
```

Enable/Disable Non maskable Pin interrupt.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Non maskable pin interrupt. true - enable Non-maskable pin interrupt. false - disable Non-maskable pin interrupt.

```
static inline uint8_t CMC_GetISPMODEPinLogic(CMC_Type *base)
```

Gets the logic state of the ISPMODE_n pin.

This function returns the logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

Parameters

- base – CMC peripheral base address.

Returns

The logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

```
static inline void CMC_ClearISPMODEPinLogic(CMC_Type *base)
```

Clears ISPMODE_n pin state.

Parameters

- base – CMC peripheral base address.

```
static inline void CMC_ForceBootConfiguration(CMC_Type *base, bool assert)
```

Set the logic state of the BOOT_CONFIGn pin.

This function force the logic state of the Boot_Confign pin to assert on next system reset.

Parameters

- base – CMC peripheral base address.
- assert – Assert the corresponding pin or not. true - Assert corresponding pin on next system reset. false - No effect.

```
static inline void CMC_LockWriteOperationToBootRomStatusReg(CMC_Type *base, uint8_t index)
```

Lock write operation to BootROM status register and BootROM Lock register.

Note: If locked, BootROM status register cannot be written.

Note: Once locked, only cold reset can reset related register.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

static inline bool CMC_CheckBootRomStatusRegWriteLocked(CMC_Type *base, uint8_t index)

Check if BootROM status register can be written.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Return values

- true – The selected BootRom status register is locked and cannot be written.
- false – The selected BootRom Status register is unlocked and cannot be written.

static inline uint32_t CMC_GetBootRomStatus(CMC_Type *base, uint8_t index)

Gets the information written by the BootROM.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Returns

The status information written by the BootROM.

static inline void CMC_WriteBootRomStatusReg(CMC_Type *base, uint8_t index, uint32_t value)

Writes value to BootROM status register, in this way, BootROM status registers are used as general purpose register.

Note: Value in BootROM status registers are reset in cold reset.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.
- value – Value to write.

void CMC_PowerOffSRAMAllMode(CMC_Type *base, uint32_t mask)

Power off the selected system SRAM always.

This function power off the selected system SRAM always. The SRAM arrays should not be accessed while they are shut down. SRAM array contents are not retained if they are powered off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered off all modes. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
static inline void CMC_PowerOnSRAMAllMode(CMC_Type *base, uint32_t mask)
```

Power on SRAM during all mode.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered on all modes. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
void CMC_PowerOffSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)
```

Power off the selected system SRAM during low power mode only.

This function power off the selected system SRAM only during low power mode. SRAM array contents are not retained if they are power off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power off during low power mode only. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
static inline void CMC_PowerOnSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)
```

Power on the selected system SRAM during low power mode only.

This function power on the selected system SRAM. The SRAM array contents are retained in low power modes.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power on during low power mode only. See `_cmc_system_sram_arrays` for details. Check Reference Manual for the SRAM region and mask bit relationship.

```
void CMC_ConfigFlashMode(CMC_Type *base, bool wake, bool doze, bool disable)
```

Configs the low power mode of the on-chip flash memory.

This function configs the low power mode of the on-chip flash memory.

Parameters

- base – CMC peripheral base address.
- wake – true: Flash will exit low power state during the flash memory accesses. false: No effect.
- doze – true: Flash is disabled while core is sleeping false: No effect.
- disable – true: Flash memory is placed in low power state. false: No effect.

```
static inline void CMC_EnableDebugOperation(CMC_Type *base, bool enable)
```

Enables/Disables debug Operation when the core sleep.

This function configs what happens to debug when core sleeps.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Debug when Core is sleeping. true - Debug remains enabled when the core is sleeping. false - Debug is disabled when the core is sleeping.

`void CMC_PreEnterLowPowerMode(void)`

Prepares to enter low power modes.

This function should be called before entering low power modes.

`void CMC_PostExitLowPowerMode(void)`

Recovers after wake up from low power modes.

This function should be called after wake up from low power modes. This function should be used with `CMC_PreEnterLowPowerMode()`

`void CMC_GlobalEnterLowPowerMode(CMC_Type *base, cmc_low_power_mode_t lowPowerMode)`

Configures the entry into the same low power mode for each power domains.

This function provides the feature to entry into the same low power mode for each power domains. Before invoking this function, please ensure the selected power mode have been allowed.

Parameters

- `base` – CMC peripheral base address.
- `lowPowerMode` – The low power mode to be entered. See `cmc_low_power_mode_t` for the details.

`void CMC_EnterLowPowerMode(CMC_Type *base, const cmc_power_domain_config_t *config)`

Configures the entry into different low power modes for each power domains.

This function provides the feature to entry into different low power modes for each power domains. Before invoking this function please ensure the selected modes are allowed.

Parameters

- `base` – CMC peripheral base address.
- `config` – Pointer to the `cmc_power_domain_config_t` structure.

`FSL_CMC_DRIVER_VERSION`

CMC driver version 2.4.3.

`CMC_SRAM_BUSY_TIMEOUT`

Max loops to wait for CMC SRAM operation complete.

When configuring the SRAM, driver will wait for the completion of new settings. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

`enum _cmc_power_mode_protection`

CMC power mode Protection enumeration.

Values:

enumerator `kCMC_AllowSleepMode`

Allow Sleep mode.

enumerator `kCMC_AllowDeepSleepMode`

Allow Deep Sleep mode.

enumerator `kCMC_AllowPowerDownMode`

Allow Power Down mode.

enumerator `kCMC_AllowDeepPowerDownMode`

Allow Deep Power Down mode.

enumerator kCMC_AllowAllLowPowerModes

Allow all low power modes.

enum _cmc_wakeup_sources

Wake up sources from the previous low power mode entry.

Values:

enumerator kCMC_WakeupFromResetInterruptOrPowerDown

Wakeup source is reset interrupt, or wake up from [Deep] Power Down.

enumerator kCMC_WakeupFromDebugRequest

Wakeup source is debug request.

enumerator kCMC_WakeupFromInterrupt

Wakeup source is interrupt.

enumerator kCMC_WakeupFromDMAWakeup

Wakeup source is DMA Wakeup.

enumerator kCMC_WakeupFromWUURequest

Wakeup source is WUU request.

enumerator kCMC_WakeupFromBusMaster

Wakeup source is Bus master.

enum _cmc_system_reset_interrupt_enable

System Reset Interrupt enable enumeration.

Values:

enumerator kCMC_PinResetInterruptEnable

Pin Reset interrupt enable.

enumerator kCMC_DAPResetInterruptEnable

DAP Reset interrupt enable.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetInterruptEnable

Low Power Acknowledge Timeout Reset interrupt enable.

enumerator kCMC_Watchdog0ResetInterruptEnable

Watchdog 0 Reset interrupt enable.

enumerator kCMC_SoftwareResetInterruptEnable

Software Reset interrupt enable.

enumerator kCMC_LockupResetInterruptEnable

Lockup Reset interrupt enable.

enumerator kCMC_Watchdog1ResetInterruptEnable

Watchdog 1 Reset interrupt enable

enum _cmc_system_reset_interrupt_flag

CMC System Reset Interrupt Status flag.

Values:

enumerator kCMC_PinResetInterruptFlag

Pin Reset interrupt flag.

enumerator kCMC_DAPResetInterruptFlag

DAP Reset interrupt flag.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetFlag
Low Power Acknowledge Timeout Reset interrupt flag.

enumerator kCMC_Watchdog0ResetInterruptFlag
Watchdog 0 Reset interrupt flag.

enumerator kCMC_SoftwareResetInterruptFlag
Software Reset interrupt flag.

enumerator kCMC_LockupResetInterruptFlag
Lock up Reset interrupt flag.

enumerator kCMC_Watchdog1ResetInterruptFlag
Watchdog 1 Reset interrupt flag.

enum _cmc_system_sram_arrays
CMC System SRAM arrays low power mode enable enumeration.

Values:

enumerator kCMC_SRAMBank0
Power off SRAM Bank0, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank1
Power off SRAM Bank1, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank2
Power off SRAM Bank2, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank3
Power off SRAM Bank3, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank4
Power off SRAM Bank4, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank5
Power off SRAM Bank5, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank6
Power off SRAM Bank6, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank7
Power off SRAM Bank7, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank8
Power off SRAM Bank8, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank9
Power off SRAM Bank9, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank10
Power off SRAM Bank10, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_AllSramArrays
Mask of all system SRAM arrays.

enum _cmc_system_reset_sources
System reset sources enumeration.

Values:

enumerator kCMC_WakeUpReset
The reset caused by a wakeup from Power Down or Deep Power Down mode.

enumerator kCMC_PORReset

The reset caused by power on reset detection logic.

enumerator kCMC_LVDRReset

The reset caused by a Low Voltage Detect.

enumerator kCMC_HVDRReset

The reset caused by a High voltage Detect.

enumerator kCMC_WarmReset

The last reset source is a warm reset source.

enumerator kCMC_FatalReset

The last reset source is a fatal reset source.

enumerator kCMC_PinReset

The reset caused by the RESET_b pin.

enumerator kCMC_DAPReset

The reset caused by a reset request from the Debug Access port.

enumerator kCMC_ResetTimeout

The reset caused by a timeout or other error condition in the system reset generation.

enumerator kCMC_LowPowerAcknowledgeTimeoutReset

The reset caused by a timeout in low power mode entry logic.

enumerator kCMC_SCGRReset

The reset caused by a loss of clock or loss of lock event in the SCG.

enumerator kCMC_Watchdog0Reset

The reset caused by a WatchDog 0 timeout.

enumerator kCMC_SoftwareReset

The reset caused by a software reset request.

enumerator kCMC_LockUpReset

The reset caused by the ARM core indication of a LOCKUP event.

enumerator kCMC_Watchdog1Reset

The reset caused by a WatchDog 1 timeout.

enum _cmc_core_clock_gate_status

Indicate the core clock was gated.

Values:

enumerator kCMC_CoreClockNotGated

Core clock not gated.

enumerator kCMC_CoreClockGated

Core clock was gated due to low power mode entry.

enum _cmc_clock_mode

CMC clock mode enumeration.

Values:

enumerator kCMC_GateNoneClock

No clock gating.

enumerator kCMC_GateCoreClock

Gate Core clock.

enumerator kCMC_GateCorePlatformClock

Gate Core clock and platform clock.

enumerator kCMC_GateAllSystemClocks

Gate all System clocks, without getting core entering into low power mode.

enumerator kCMC_GateAllSystemClocksEnterLowPowerMode

Gate all System clocks, with core entering into low power mode.

enum _cmc_low_power_mode

CMC power mode enumeration.

Values:

enumerator kCMC_ActiveMode

Select Active mode.

enumerator kCMC_SleepMode

Select Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepSleepMode

Select Deep Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_PowerDownMode

Select Power Down mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepPowerDown

Select Deep Power Down mode when a core executes WFI or WFE instruction.

typedef enum _cmc_core_clock_gate_status cmc_core_clock_gate_status_t

Indicate the core clock was gated.

typedef enum _cmc_clock_mode cmc_clock_mode_t

CMC clock mode enumeration.

typedef enum _cmc_low_power_mode cmc_low_power_mode_t

CMC power mode enumeration.

typedef struct _cmc_reset_pin_config cmc_reset_pin_config_t

CMC reset pin configuration.

typedef struct _cmc_power_domain_config cmc_power_domain_config_t

power mode configuration for each power domain.

CMC_BLR_LOCK_FIELD_WIDTH

CMC_BLR_LOCK_IDX_MASK(index)

CMC_BLR_LOCK_IDX_SHIFT(index)

CMC_BLR_LOCK_IDX(index, value)

struct _cmc_reset_pin_config

#include <fsl_cmc.h> CMC reset pin configuration.

Public Members

bool lowpowerFilterEnable

Low Power Filter enable.

bool resetFilterEnable

Reset Filter enable.

uint8_t resetFilterWidth

Width of the Reset Filter.

struct __cmc_power_domain_config

#include <fsl_cmc.h> power mode configuration for each power domain.

Public Members

cmc_clock_mode_t clock_mode

Clock mode for each power domain.

cmc_low_power_mode_t main_domain

The low power mode of the MAIN power domain.

cmc_low_power_mode_t wake_domain

The low power mode of the WAKE power domain.

2.9 MIPI CSI2 RX: MIPI CSI2 RX Driver

FSL_CSI2RX_DRIVER_VERSION

CSI2RX driver version.

enum __csi2rx_data_lane

CSI2RX data lanes.

Values:

enumerator kCSI2RX_DataLane0

Data lane 0.

enumerator kCSI2RX_DataLane1

Data lane 1.

enumerator kCSI2RX_DataLane2

Data lane 2.

enumerator kCSI2RX_DataLane3

Data lane 3.

enum __csi2rx_payload

CSI2RX payload type.

Values:

enumerator kCSI2RX_PayloadGroup0Null

NULL.

enumerator kCSI2RX_PayloadGroup0Blank

Blank.

enumerator kCSI2RX_PayloadGroup0Embedded

Embedded.

enumerator kCSI2RX_PayloadGroup0YUV420_8Bit

Legacy YUV420 8 bit.

enumerator kCSI2RX_PayloadGroup0YUV422_8Bit

YUV422 8 bit.

enumerator kCSI2RX_PayloadGroup0YUV422_10Bit
YUV422 10 bit.

enumerator kCSI2RX_PayloadGroup0RGB444
RGB444.

enumerator kCSI2RX_PayloadGroup0RGB555
RGB555.

enumerator kCSI2RX_PayloadGroup0RGB565
RGB565.

enumerator kCSI2RX_PayloadGroup0RGB666
RGB666.

enumerator kCSI2RX_PayloadGroup0RGB888
RGB888.

enumerator kCSI2RX_PayloadGroup0Raw6
Raw 6.

enumerator kCSI2RX_PayloadGroup0Raw7
Raw 7.

enumerator kCSI2RX_PayloadGroup0Raw8
Raw 8.

enumerator kCSI2RX_PayloadGroup0Raw10
Raw 10.

enumerator kCSI2RX_PayloadGroup0Raw12
Raw 12.

enumerator kCSI2RX_PayloadGroup0Raw14
Raw 14.

enumerator kCSI2RX_PayloadGroup1UserDefined1
User defined 8-bit data type 1, 0x30.

enumerator kCSI2RX_PayloadGroup1UserDefined2
User defined 8-bit data type 2, 0x31.

enumerator kCSI2RX_PayloadGroup1UserDefined3
User defined 8-bit data type 3, 0x32.

enumerator kCSI2RX_PayloadGroup1UserDefined4
User defined 8-bit data type 4, 0x33.

enumerator kCSI2RX_PayloadGroup1UserDefined5
User defined 8-bit data type 5, 0x34.

enumerator kCSI2RX_PayloadGroup1UserDefined6
User defined 8-bit data type 6, 0x35.

enumerator kCSI2RX_PayloadGroup1UserDefined7
User defined 8-bit data type 7, 0x36.

enumerator kCSI2RX_PayloadGroup1UserDefined8
User defined 8-bit data type 8, 0x37.

enum _csi2rx_bit_error
MIPI CSI2RX bit errors.

Values:

enumerator kCSI2RX_BitErrorEccTwoBit
ECC two bit error has occurred.

enumerator kCSI2RX_BitErrorEccOneBit
ECC one bit error has occurred.

enum _csi2rx_ppi_error
MIPI CSI2RX PPI error types.

Values:

enumerator kCSI2RX_PpiErrorSotHs
CSI2RX DPHY PPI error ErrSotHS.

enumerator kCSI2RX_PpiErrorSotSyncHs
CSI2RX DPHY PPI error ErrSotSync_HS.

enumerator kCSI2RX_PpiErrorEsc
CSI2RX DPHY PPI error ErrEsc.

enumerator kCSI2RX_PpiErrorSyncEsc
CSI2RX DPHY PPI error ErrSyncEsc.

enumerator kCSI2RX_PpiErrorControl
CSI2RX DPHY PPI error ErrControl.

enum _csi2rx_interrupt
MIPI CSI2RX interrupt.

Values:

enumerator kCSI2RX_InterruptCrcError

enumerator kCSI2RX_InterruptEccOneBitError

enumerator kCSI2RX_InterruptEccTwoBitError

enumerator kCSI2RX_InterruptUlpsStatusChange

enumerator kCSI2RX_InterruptErrorSotHs

enumerator kCSI2RX_InterruptErrorSotSyncHs

enumerator kCSI2RX_InterruptErrorEsc

enumerator kCSI2RX_InterruptErrorSyncEsc

enumerator kCSI2RX_InterruptErrorControl

enum _csi2rx_ulps_status
MIPI CSI2RX D-PHY ULPS state.

Values:

enumerator kCSI2RX_ClockLaneUlps
Clock lane is in ULPS state.

enumerator kCSI2RX_DataLane0Ulps
Data lane 0 is in ULPS state.

enumerator kCSI2RX_DataLane1Ulps
Data lane 1 is in ULPS state.

enumerator kCSI2RX_DataLane2Ulps
Data lane 2 is in ULPS state.

enumerator kCSI2RX_DataLane3Ulps

Data lane 3 is in ULPS state.

enumerator kCSI2RX_ClockLaneMark

Clock lane is in mark state.

enumerator kCSI2RX_DataLane0Mark

Data lane 0 is in mark state.

enumerator kCSI2RX_DataLane1Mark

Data lane 1 is in mark state.

enumerator kCSI2RX_DataLane2Mark

Data lane 2 is in mark state.

enumerator kCSI2RX_DataLane3Mark

Data lane 3 is in mark state.

typedef struct *_csi2rx_config* csi2rx_config_t

CSI2RX configuration.

typedef enum *_csi2rx_ppi_error* csi2rx_ppi_error_t

MIPI CSI2RX PPI error types.

void CSI2RX_Init(MIPI_CSI2RX_Type *base, const *csi2rx_config_t* *config)

Enables and configures the CSI2RX peripheral module.

Parameters

- base – CSI2RX peripheral address.
- config – CSI2RX module configuration structure.

void CSI2RX_Deinit(MIPI_CSI2RX_Type *base)

Disables the CSI2RX peripheral module.

Parameters

- base – CSI2RX peripheral address.

static inline uint32_t CSI2RX_GetBitError(MIPI_CSI2RX_Type *base)

Gets the MIPI CSI2RX bit error status.

This function gets the RX bit error status, the return value could be compared with *_csi2rx_bit_error*. If one bit ECC error detected, the return value could be passed to the function *CSI2RX_GetEccBitErrorPosition* to get the position of the ECC error bit.

Example:

```
uint32_t bitError;
uint32_t bitErrorPosition;

bitError = CSI2RX_GetBitError(MIPI_CSI2RX);

if (kCSI2RX_BitErrorEccTwoBit & bitError)
{
    Two bits error;
}
else if (kCSI2RX_BitErrorEccOneBit & bitError)
{
    One bits error;
    bitErrorPosition = CSI2RX_GetEccBitErrorPosition(bitError);
}
```

Parameters

- base – CSI2RX peripheral address.

Returns

The RX bit error status.

```
static inline uint32_t CSI2RX_GetEccBitErrorPosition(uint32_t bitError)
```

Get ECC one bit error bit position.

If CSI2RX_GetBitError detects ECC one bit error, this function could extract the error bit position from the return value of CSI2RX_GetBitError.

Parameters

- bitError – The bit error returned by CSI2RX_GetBitError.

Returns

The position of error bit.

```
static inline uint32_t CSI2RX_GetUlpsStatus(MIPI_CSI2RX_Type *base)
```

Gets the MIPI CSI2RX D-PHY ULPS status.

Example to check whether data lane 0 is in ULPS status.

```
uint32_t status = CSI2RX_GetUlpsStatus(MIPI_CSI2RX);

if (kCSI2RX_DataLane0Ulps & status)
{
    Data lane 0 is in ULPS status.
}
```

Parameters

- base – CSI2RX peripheral address.

Returns

The MIPI CSI2RX D-PHY ULPS status, it is OR'ed value or _csi2rx_ulps_status.

```
static inline uint32_t CSI2RX_GetPpiErrorDataLanes(MIPI_CSI2RX_Type *base,
                                                    csi2rx_ppi_error_t errorType)
```

Gets the MIPI CSI2RX D-PHY PPI error lanes.

This function checks the PPI error occurred on which data lanes, the returned value is OR'ed value of csi2rx_ppi_error_t. For example, if the ErrSotHS is detected, to check the ErrSotHS occurred on which data lanes, use like this:

```
uint32_t errorDataLanes = CSI2RX_GetPpiErrorDataLanes(MIPI_CSI2RX, kCSI2RX_
↪PpiErrorSotHS);

if (kCSI2RX_DataLane0 & errorDataLanes)
{
    ErrSotHS occurred on data lane 0.
}

if (kCSI2RX_DataLane1 & errorDataLanes)
{
    ErrSotHS occurred on data lane 1.
}
```

Parameters

- base – CSI2RX peripheral address.
- errorType – What kind of error to check.

Returns

The data lane mask that error errorType occurred.

static inline void CSI2RX_EnableInterrupts(MIPI_CSI2RX_Type *base, uint32_t mask)

Enable the MIPI CSI2RX interrupts.

This function enables the MIPI CSI2RX interrupts. The interrupts to enable are passed in as an OR'ed value of `_csi2rx_interrupt`. For example, to enable one bit and two bit ECC error interrupts, use like this:

```
CSI2RX_EnableInterrupts(MIPI_CSI2RX, kCSI2RX_InterruptEccOneBitError | kCSI2RX_InterruptEccTwoBitError);
```

Parameters

- base – CSI2RX peripheral address.
- mask – OR'ed value of `_csi2rx_interrupt`.

static inline void CSI2RX_DisableInterrupts(MIPI_CSI2RX_Type *base, uint32_t mask)

Disable the MIPI CSI2RX interrupts.

This function disables the MIPI CSI2RX interrupts. The interrupts to disable are passed in as an OR'ed value of `_csi2rx_interrupt`. For example, to disable one bit and two bit ECC error interrupts, use like this:

```
CSI2RX_DisableInterrupts(MIPI_CSI2RX, kCSI2RX_InterruptEccOneBitError | kCSI2RX_InterruptEccTwoBitError);
```

Parameters

- base – CSI2RX peripheral address.
- mask – OR'ed value of `_csi2rx_interrupt`.

static inline uint32_t CSI2RX_GetInterruptStatus(MIPI_CSI2RX_Type *base)

Get the MIPI CSI2RX interrupt status.

This function returns the MIPI CSI2RX interrupts status as an OR'ed value of `_csi2rx_interrupt`.

Parameters

- base – CSI2RX peripheral address.

Returns

OR'ed value of `_csi2rx_interrupt`.

CSI2RX_REG_CFG_NUM_LANES(base)

CSI2RX_REG_CFG_DISABLE_DATA_LANES(base)

CSI2RX_REG_BIT_ERR(base)

CSI2RX_REG_IRQ_STATUS(base)

CSI2RX_REG_IRQ_MASK(base)

CSI2RX_REG_ULPS_STATUS(base)

CSI2RX_REG_PPI_ERRSOT_HS(base)

CSI2RX_REG_PPI_ERRSOTSYNC_HS(base)

CSI2RX_REG_PPI_ERRESC(base)

CSI2RX_REG_PPI_ERRSYNCESC(base)

```

CSI2RX_REG_PPI_ERRCONTROL(base)
CSI2RX_REG_CFG_DISABLE_PAYLOAD_0(base)
CSI2RX_REG_CFG_DISABLE_PAYLOAD_1(base)
CSI2RX_REG_CFG_IGNORE_VC(base)
CSI2RX_REG_CFG_VID_VC(base)
CSI2RX_REG_CFG_VID_P_FIFO_SEND_LEVEL(base)
CSI2RX_REG_CFG_VID_VSYNC(base)
CSI2RX_REG_CFG_VID_HSYNC_FP(base)
CSI2RX_REG_CFG_VID_HSYNC(base)
CSI2RX_REG_CFG_VID_HSYNC_BP(base)
MIPI_CSI2RX_CSI2RX_CFG_NUM_LANES_csi2rx_cfg_num_lanes_MASK
MIPI_CSI2RX_CSI2RX_IRQ_MASK_csi2rx_irq_mask_MASK
struct _csi2rx_config
    #include <fsl_mipi_csi2rx.h> CSI2RX configuration.

```

Public Members

```

uint8_t laneNum
    Number of active lanes used for receiving data.

uint8_t tHsSettle_EscClk
    Number of rx_clk_esc clock periods for T_HS_SETTLE. The T_HS_SETTLE should be in
    the range of 85ns + 6UI to 145ns + 10UI.

```

2.10 DAC12: 12-bit Digital-to-Analog Converter Driver

```

void DAC12_GetHardwareInfo(DAC_Type *base, dac12_hardware_info_t *info)
    Get hardware information about this module.

```

Parameters

- base – DAC12 peripheral base address.
- info – Pointer to info structure, see to dac12_hardware_info_t.

```

void DAC12_Init(DAC_Type *base, const dac12_config_t *config)
    Initialize the DAC12 module.

```

Parameters

- base – DAC12 peripheral base address.
- config – Pointer to configuration structure, see to dac12_config_t.

```

void DAC12_GetDefaultConfig(dac12_config_t *config)
    Initializes the DAC12 user configuration structure.

    This function initializes the user configuration structure to a default value. The default
    values are:

```

```
config->fifoWatermarkLevel = 0U;  
config->fifoWorkMode = kDAC12_FIFODisabled;  
config->referenceVoltageSource = kDAC12_ReferenceVoltageSourceAlt1;  
config->fifoTriggerMode = kDAC12_FIFOTriggerByHardwareMode;  
config->referenceCurrentSource = kDAC12_ReferenceCurrentSourceAlt0;  
config->speedMode = kDAC12_SpeedLowMode;  
config->speedMode = false;  
config->currentReferenceInternalTrimValue = 0x4;
```

Parameters

- config – Pointer to the configuration structure. See “dac12_config_t”.

void DAC12_Deinit(DAC_Type *base)

De-initialize the DAC12 module.

Parameters

- base – DAC12 peripheral base address.

static inline void DAC12_Enable(DAC_Type *base, bool enable)

Enable the DAC12's converter or not.

Parameters

- base – DAC12 peripheral base address.
- enable – Enable the DAC12's converter or not.

static inline void DAC12_ResetConfig(DAC_Type *base)

Reset all internal logic and registers.

Parameters

- base – DAC12 peripheral base address.

static inline void DAC12_ResetFIFO(DAC_Type *base)

Reset the FIFO pointers.

FIFO pointers should only be reset when the DAC12 is disabled. This function can be used to configure both pointers to the same address to reset the FIFO as empty.

Parameters

- base – DAC12 peripheral base address.

static inline uint32_t DAC12_GetStatusFlags(DAC_Type *base)

Get status flags.

Parameters

- base – DAC12 peripheral base address.

Returns

Mask of current status flags. See to _dac12_status_flags.

static inline void DAC12_ClearStatusFlags(DAC_Type *base, uint32_t flags)

Clear status flags.

Note: Not all the flags can be cleared by this API. Several flags need special condition to clear them according to target chip's reference manual document.

Parameters

- base – DAC12 peripheral base address.
- flags – Mask of status flags to be cleared. See to _dac12_status_flags.

static inline void DAC12_EnableInterrupts(DAC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – DAC12 peripheral base address.
- mask – Mask value of interrupts to be enabled. See to `_dac12_interrupt_enable`.

static inline void DAC12_DisableInterrupts(DAC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – DAC12 peripheral base address.
- mask – Mask value of interrupts to be disabled. See to `_dac12_interrupt_enable`.

static inline void DAC12_EnableDMA(DAC_Type *base, bool enable)

Enable DMA or not.

When DMA is enabled, the DMA request will be generated by original interrupts. The interrupts will not be presented on this module at the same time.

static inline void DAC12_SetData(DAC_Type *base, uint32_t value)

Set data into the entry of FIFO buffer.

When the DAC FIFO is disabled, and the one entry buffer is enabled, the DAC converts the data in the buffer to analog output voltage. Any write to the DATA register will replace the data in the buffer and push data to analog conversion without trigger support. When the DAC FIFO is enabled. Writing data would increase the write pointer of FIFO. Also, the data would be restored into the FIFO buffer.

Parameters

- base – DAC12 peripheral base address.
- value – Setting value into FIFO buffer.

static inline void DAC12_DoSoftwareTrigger(DAC_Type *base)

Do trigger the FIFO by software.

When the DAC FIFO is enabled, and software trigger is used. Doing trigger would increase the read pointer, and the data in the entry pointed by read pointer would be converted as new output.

Parameters

- base – DAC12 peripheral base address.

static inline uint32_t DAC12_GetFIFOReadPointer(DAC_Type *base)

Get the current read pointer of FIFO.

Parameters

- base – DAC12 peripheral base address.

Returns

Read pointer index of FIFO buffer.

static inline uint32_t DAC12_GetFIFOWritePointer(DAC_Type *base)

Get the current write pointer of FIFO.

Parameters

- base – DAC12 peripheral base address.

Returns

Write pointer index of FIFO buffer

FSL_DAC12_DRIVER_VERSION

DAC12 driver version 2.1.2.

enum __dac12_status_flags

DAC12 flags.

Values:

enumerator kDAC12_OverflowFlag

FIFO overflow status flag, which indicates that more data has been written into FIFO than it can hold.

enumerator kDAC12_UnderflowFlag

FIFO underflow status flag, which means that there is a new trigger after the FIFO is nearly empty.

enumerator kDAC12_WatermarkFlag

FIFO watermark status flag, which indicates the remaining FIFO data is less than the watermark setting.

enumerator kDAC12_NearlyEmptyFlag

FIFO nearly empty flag, which means there is only one data remaining in FIFO.

enumerator kDAC12_FullFlag

FIFO full status flag, which means that the FIFO read pointer equals the write pointer, as the write pointer increase.

enum __dac12_interrupt_enable

DAC12 interrupts.

Values:

enumerator kDAC12_UnderOrOverflowInterruptEnable

Underflow and overflow interrupt enable.

enumerator kDAC12_WatermarkInterruptEnable

Watermark interrupt enable.

enumerator kDAC12_NearlyEmptyInterruptEnable

Nearly empty interrupt enable.

enumerator kDAC12_FullInterruptEnable

Full interrupt enable.

enum __dac12_fifo_size_info

DAC12 FIFO size information provided by hardware.

Values:

enumerator kDAC12_FIFOSize2

FIFO depth is 2.

enumerator kDAC12_FIFOSize4

FIFO depth is 4.

enumerator kDAC12_FIFOSize8

FIFO depth is 8.

enumerator kDAC12_FIFOSize16

FIFO depth is 16.

enumerator kDAC12__FIFOSize32

FIFO depth is 32.

enumerator kDAC12__FIFOSize64

FIFO depth is 64.

enumerator kDAC12__FIFOSize128

FIFO depth is 128.

enumerator kDAC12__FIFOSize256

FIFO depth is 256.

enum __dac12_fifo_work_mode

DAC12 FIFO work mode.

Values:

enumerator kDAC12__FIFODisabled

FIFO disabled and only one level buffer is enabled. Any data written from this buffer goes to conversion.

enumerator kDAC12__FIFOWorkAsNormalMode

Data will first read from FIFO to buffer then go to conversion.

enumerator kDAC12__FIFOWorkAsSwingMode

In Swing mode, the FIFO must be set up to be full. In Swing back mode, a trigger changes the read pointer to make it swing between the FIFO Full and Nearly Empty state. That is, the trigger increases the read pointer till FIFO is nearly empty and decreases the read pointer till the FIFO is full.

enum __dac12_reference_voltage_source

DAC12 reference voltage source.

Values:

enumerator kDAC12__ReferenceVoltageSourceAlt1

The DAC selects DACREF_1 as the reference voltage.

enumerator kDAC12__ReferenceVoltageSourceAlt2

The DAC selects DACREF_2 as the reference voltage.

enum __dac12_fifo_trigger_mode

DAC12 FIFO trigger mode.

Values:

enumerator kDAC12__FIFOTriggerByHardwareMode

Buffer would be triggered by hardware.

enumerator kDAC12__FIFOTriggerBySoftwareMode

Buffer would be triggered by software.

enum __dac12_reference_current_source

DAC internal reference current source.

Analog module needs reference current to keep working . Such reference current can generated by IP itself, or by on-chip PMC's "reference part". If no current reference be selected, analog module can't working normally ,even when other register can still be assigned, DAC would waste current but no function. To make the DAC work, either kDAC12__ReferenceCurrentSourceAltX should be selected.

Values:

enumerator kDAC12_ReferenceCurrentSourceDisabled

None of reference current source is enabled.

enumerator kDAC12_ReferenceCurrentSourceAlt0

Use the internal reference current generated by the module itself.

enumerator kDAC12_ReferenceCurrentSourceAlt1

Use the ZTC(Zero Temperature Coefficient) reference current generated by on-chip power management module.

enumerator kDAC12_ReferenceCurrentSourceAlt2

Use the PTAT(Proportional To Absolution Temperature) reference current generated by power management module.

enum _dac12_speed_mode

DAC analog buffer speed mode for conversion.

Values:

enumerator kDAC12_SpeedLowMode

Low speed mode.

enumerator kDAC12_SpeedMiddleMode

Middle speed mode.

enumerator kDAC12_SpeedHighMode

High speed mode.

typedef enum _dac12_fifo_size_info dac12_fifo_size_info_t

DAC12 FIFO size information provided by hardware.

typedef enum _dac12_fifo_work_mode dac12_fifo_work_mode_t

DAC12 FIFO work mode.

typedef enum _dac12_reference_voltage_source dac12_reference_voltage_source_t

DAC12 reference voltage source.

typedef enum _dac12_fifo_trigger_mode dac12_fifo_trigger_mode_t

DAC12 FIFO trigger mode.

typedef enum _dac12_reference_current_source dac12_reference_current_source_t

DAC internal reference current source.

Analog module needs reference current to keep working . Such reference current can generated by IP itself, or by on-chip PMC's "reference part". If no current reference be selected, analog module can't working normally ,even when other register can still be assigned, DAC would waste current but no function. To make the DAC work, either kDAC12_ReferenceCurrentSourceAltx should be selected.

typedef enum _dac12_speed_mode dac12_speed_mode_t

DAC analog buffer speed mode for conversion.

typedef struct _dac12_hardware_info dac12_hardware_info_t

DAC12 hardware information.

DAC12_CR_W1C_FLAGS_MASK

Define "write 1 to clear" flags.

DAC12_CR_ALL_FLAGS_MASK

Define all the flag bits in DACx_CR register.

struct _dac12_hardware_info

#include <fsl_dac12.h> DAC12 hardware information.

Public Members

dac12_fifo_size_info_t fifoSizeInfo

The number of words in this device's DAC buffer.

struct *dac12_config_t*

#include <fsl_dac12.h> DAC12 module configuration.

Actually, the most fields are for FIFO buffer.

Public Members

uint32_t fifoWatermarkLevel

FIFO's watermark, the max value can be the hardware FIFO size.

dac12_fifo_work_mode_t fifoWorkMode

FIFO's work mode about pointers.

dac12_reference_voltage_source_t referenceVoltageSource

Select the reference voltage source.

dac12_reference_current_source_t referenceCurrentSource

Select the trigger mode for FIFO. Select the reference current source.

dac12_speed_mode_t speedMode

Select the speed mode for conversion.

bool enableAnalogBuffer

Enable analog buffer for high drive.

2.11 EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

void EDMA_AD_Init(DMA_AD_Type *base, const *edma_config_t* *config)

Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Note: This function enables the minor loop map feature.

Parameters

- base – eDMA peripheral base address.
- config – A pointer to the configuration structure, see “*edma_config_t*”.

void EDMA_AD_Deinit(DMA_AD_Type *base)

Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- base – eDMA peripheral base address.

```
void EDMA_AD_InstallTCD(DMA_AD_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_AD_GetDefaultConfig(edma_config_t *config)
```

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values:

```
config.enableMasterIdReplication = true;  
config.enableHaltOnError = true;  
config.enableRoundRobinArbitration = false;  
config.enableDebugMode = false;  
config.enableBufferedWrites = false;
```

Parameters

- config – A pointer to the eDMA configuration structure.

```
static inline void EDMA_AD_EnableAllChannelLink(DMA_AD_Type *base, bool enable)
```

Enables/disables all channel linking.

This function enables/disables all channel linking in the management page. For specific channel linking enablement & configuration, please refer to `EDMA_SetChannelLink` and `EDMA_TcdSetChannelLink` APIs.

For example, to disable all channel linking in the DMA0 management page:

```
EDMA_EnableAllChannelLink(DMA0, false);
```

Parameters

- base – eDMA peripheral base address.
- enable – Switcher of the channel linking feature for all channels. “true” means to enable. “false” means not.

```
void EDMA_AD_ResetChannel(DMA_AD_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_AD_SetTransferConfig(DMA_AD_Type *base, uint32_t channel, const
                             edma_transfer_config_t *config, edma_tcd_t *nextTcd)
```

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_config_t config;
edma_tcd_t tcd;
config.srcAddr = ..;
config.destAddr = ..;
...
EDMA_AD_SetTransferConfig(DMA0, channel, &config, &tcd);
```

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_AD_SetMinorOffsetConfig(DMA_AD_Type *base, uint32_t channel, const
                                  edma_minor_offset_config_t *config)
```

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – A pointer to the minor offset configuration structure.

```
static inline void EDMA_AD_SetChannelArbitrationGroup(DMA_AD_Type *base, uint32_t
                                                       channel, uint32_t group)
```

Configures the eDMA channel arbitration group.

This function configures the channel arbitration group. The arbitration group priorities are evaluated by numeric value from highest group number to lowest.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- group – Fixed-priority arbitration group number for the channel.

```
static inline void EDMA_AD_SetChannelPreemptionConfig(DMA_AD_Type *base, uint32_t
                                                       channel, const
                                                       edma_channel_Preemption_config_t
                                                       *config)
```

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- config – A pointer to the channel preemption configuration structure.

```
static inline uint32_t EDMA_AD_GetChannelSystemBusInformation(DMA_AD_Type *base,  
                                                             uint32_t channel)
```

Gets the eDMA channel identification and attribute information on the system bus interface.

Parameters

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

Returns

The mask of the channel system bus information. Users need to use the `_edma_channel_sys_bus_info` type to decode the return variables.

```
void EDMA_AD_SetChannelLink(DMA_AD_Type *base, uint32_t channel,  
                             edma_channel_link_type_t type, uint32_t linkedChannel)
```

Sets the channel link for the eDMA transfer.

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- type – A channel link type, which can be one of the following:
 - kEDMA_LinkNone
 - kEDMA_MinorLink
 - kEDMA_MajorLink
- linkedChannel – The linked channel number.

```
void EDMA_AD_SetBandWidth(DMA_AD_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_AD_SetModulo(DMA_AD_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_AD_EnableAsyncRequest(DMA_AD_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_AD_EnableAutoStopRequest(DMA_AD_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_AD_EnableChannelInterrupts(DMA_AD_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_AD_DisableChannelInterrupts(DMA_AD_Type *base, uint32_t channel, uint32_t mask)
```

Disables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of the interrupt source to be set. Use the defined `edma_interrupt_enable_t` type.

```
static inline void EDMA_AD_SetChannelMux(DMA_AD_Type *base, uint32_t channel, uint32_t mux)
```

Set channel mux source.

Note: When the peripheral is no longer needed, the mux configuration for that channel should be written to 0, thus releasing the resource.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mux – the mux source value is SOC specific, please reference the SOC for detail.

```
void EDMA_AD_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_AD_TcdSetTransferConfig(edma_tcd_t *tcd, const edma_transfer_config_t *config, edma_tcd_t *nextTcd)
```

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_config_t config = {  
    ...  
}  
edma_tcd_t tcd __aligned(32);  
edma_tcd_t nextTcd __aligned(32);  
EDMA_AD_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_AD_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_AD_TcdSetChannelLink(edma_tcd_t *tcd, edma_channel_link_type_t type, uint32_t
                              linkedChannel)
```

Sets the channel link for the eDMA TCD.

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- tcd – Point to the TCD structure.
- type – Channel link type, it can be one of:
 - kEDMA_LinkNone
 - kEDMA_MinorLink
 - kEDMA_MajorLink
- linkedChannel – The linked channel number.

```
static inline void EDMA_AD_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_AD_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_AD_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_AD_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_AD_TcdDisableInterrupts(*edma_tcd_t* *tcd, uint32_t mask)

Disables the interrupt source for the eDMA TCD.

Parameters

- tcd – Point to the TCD structure.
- mask – The mask of interrupt source to be set. Users need to use the defined *edma_interrupt_enable_t* type.

static inline void EDMA_AD_EnableChannelRequest(DMA_AD_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_AD_DisableChannelRequest(DMA_AD_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_AD_TriggerChannelStart(DMA_AD_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_AD_GetRemainingMajorLoopCount(DMA_AD_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_AD_GetErrorStatusFlags(DMA_AD_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_AD_GetChannelStatusFlags(DMA_AD_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_AD_ClearChannelStatusFlags(DMA_AD_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_AD_CreateHandle(edma_handle_t *handle, DMA_AD_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_AD_InstallTCDMemory(edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
```

Installs the TCDs memory pool into the eDMA handle.

This function is called after the `EDMA_CreateHandle` to use scatter/gather feature.

Parameters

- handle – eDMA handle pointer.
- tcdPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- tcdSize – The number of TCD slots.

```
void EDMA_AD_SetCallback(edma_handle_t *handle, edma_callback callback, void *userData)
```

Installs a callback function for the eDMA transfer.

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

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

```
void EDMA_AD_PrepareTransferConfig(edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)
```

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- `config` – The user configuration structure of type `edma_transfer_config_t`.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `srcOffset` – eDMA transfer source address offset
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `destOffset` – eDMA transfer destination address offset
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.

```
void EDMA_AD_PrepareTransfer(edma_transfer_config_t *config, void *srcAddr, uint32_t  
                             srcWidth, void *destAddr, uint32_t destWidth, uint32_t  
                             bytesEachRequest, uint32_t transferBytes,  
                             edma_transfer_type_t transferType)
```

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- `config` – The user configuration structure of type `edma_transfer_config_t`.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.
- `transferType` – eDMA transfer type.

```
status_t EDMA_AD_SubmitTransfer(edma_handle_t *handle, const edma_transfer_config_t  
                                *config)
```

Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- `handle` – eDMA handle pointer.
- `config` – Pointer to eDMA transfer configuration structure.

Return values

- `kStatus_EDMA_Success` – It means submit transfer request succeed.

- `kStatus_EDMA_QueueFull` – It means TCD queue is full. Submit transfer request is not allowed.
- `kStatus_EDMA_Busy` – It means the given channel is busy, need to submit request later.

`void EDMA_AD_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_AD_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_AD_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_AD_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_AD_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_AD_HandleIRQ(edma_handle_t *handle)`

eDMA IRQ handler for the current major loop transfer completion.

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

Parameters

- `handle` – eDMA handle pointer.

void EDMA_Init(DMA_Type *base, const *edma_config_t* *config)

Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Note: This function enables the minor loop map feature.

Parameters

- base – eDMA peripheral base address.
- config – A pointer to the configuration structure, see “*edma_config_t*”.

void EDMA_Deinit(DMA_Type *base)

Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- base – eDMA peripheral base address.

void EDMA_InstallTCD(DMA_Type *base, uint32_t channel, *edma_tcd_t* *tcd)

Push content of TCD structure into hardware TCD register.

Parameters

- base – EDMA peripheral base address.
- channel – EDMA channel number.
- tcd – Point to TCD structure.

void EDMA_GetDefaultConfig(*edma_config_t* *config)

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values:

```
config.enableMasterIdReplication = true;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
config.enableBufferedWrites = false;
```

Parameters

- config – A pointer to the eDMA configuration structure.

static inline void EDMA_EnableAllChannelLink(DMA_Type *base, bool enable)

Enables/disables all channel linking.

This function enables/disables all channel linking in the management page. For specific channel linking enablement & configuration, please refer to *EDMA_SetChannelLink* and *EDMA_TcdSetChannelLink* APIs.

For example, to disable all channel linking in the DMA0 management page:

```
EDMA_EnableAllChannelLink(DMA0, false);
```

Parameters

- base – eDMA peripheral base address.

- enable – Switcher of the channel linking feature for all channels. “true” means to enable. “false” means not.

void EDMA_ResetChannel(DMA_Type *base, uint32_t channel)

Sets all TCD registers to default values.

This function sets TCD registers for this channel to default values.

Note: This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

Note: This function enables the auto stop request feature.

Parameters

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

void EDMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *edma_transfer_config_t* *config, *edma_tcd_t* *nextTcd)

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_config_t config;
edma_tcd_t tcd;
config.srcAddr = ..;
config.destAddr = ..;
...
EDMA_SetTransferConfig(DMA0, channel, &config, &stcd);
```

Note: If nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_ResetChannel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

void EDMA_SetMinorOffsetConfig(DMA_Type *base, uint32_t channel, const *edma_minor_offset_config_t* *config)

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

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

- `config` – A pointer to the minor offset configuration structure.

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

Configures the eDMA channel arbitration group.

This function configures the channel arbitration group. The arbitration group priorities are evaluated by numeric value from highest group number to lowest.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number
- `group` – Fixed-priority arbitration group number for the channel.

```
static inline void EDMA_SetChannelPreemptionConfig(DMA_Type *base, uint32_t channel, const
                                                    edma_channel_preemption_config_t
                                                    *config)
```

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number
- `config` – A pointer to the channel preemption configuration structure.

```
static inline uint32_t EDMA_GetChannelSystemBusInformation(DMA_Type *base, uint32_t
                                                           channel)
```

Gets the eDMA channel identification and attribute information on the system bus interface.

Parameters

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

Returns

The mask of the channel system bus information. Users need to use the `_edma_channel_sys_bus_info` type to decode the return variables.

```
void EDMA_SetChannelLink(DMA_Type *base, uint32_t channel, edma_channel_link_type_t
                        type, uint32_t linkedChannel)
```

Sets the channel link for the eDMA transfer.

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.
- `type` – A channel link type, which can be one of the following:
 - `kEDMA_LinkNone`

- kEDMA_MinorLink
- kEDMA_MajorLink
- linkedChannel – The linked channel number.

void EDMA_SetBandWidth(DMA_Type *base, uint32_t channel, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA transfer.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- bandWidth – A bandwidth setting, which can be one of the following:
 - kEDMABandwidthStallNone
 - kEDMABandwidthStall4Cycle
 - kEDMABandwidthStall8Cycle

void EDMA_SetModulo(DMA_Type *base, uint32_t channel, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA transfer.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

static inline void EDMA_EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an async request for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

static inline void EDMA_EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an auto stop request for the eDMA transfer.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

`void EDMA__EnableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)`
 Enables the interrupt source for the eDMA transfer.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.
- `mask` – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

`void EDMA__DisableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)`
 Disables the interrupt source for the eDMA transfer.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.
- `mask` – The mask of the interrupt source to be set. Use the defined `edma_interrupt_enable_t` type.

`static inline void EDMA__SetChannelMux(DMA_Type *base, uint32_t channel, uint32_t mux)`
 Set channel mux source.

Note: When the peripheral is no longer needed, the mux configuration for that channel should be written to 0, thus releasing the resource.

Parameters

- `base` – eDMA peripheral base address.
- `channel` – eDMA channel number.
- `mux` – the mux source value is SOC specific, please reference the SOC for detail.

`void EDMA__TcdReset(edma_tcd_t *tcd)`
 Sets all fields to default values for the TCD structure.
 This function sets all fields for this TCD structure to default value.

Note: This function enables the auto stop request feature.

Parameters

- `tcd` – Pointer to the TCD structure.

`void EDMA__TcdSetTransferConfig(edma_tcd_t *tcd, const edma_transfer_config_t *config, edma_tcd_t *nextTcd)`

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_config_t config = {
...
}
edma_tcd_t tcd __aligned(32);
edma_tcd_t nextTcd __aligned(32);
EDMA__TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

Parameters

- tcd – Pointer to the TCD structure.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA__TcdSetMinorOffsetConfig(edma_tcd_t *tcd, const edma_minor_offset_config_t
                                   *config)
```

Configures the eDMA TCD minor offset feature.

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

- tcd – A point to the TCD structure.
- config – A pointer to the minor offset configuration structure.

```
void EDMA__TcdSetChannelLink(edma_tcd_t *tcd, edma_channel_link_type_t type, uint32_t
                             linkedChannel)
```

Sets the channel link for the eDMA TCD.

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- tcd – Point to the TCD structure.
- type – Channel link type, it can be one of:
 - kEDMA_LinkNone
 - kEDMA_MinorLink
 - kEDMA_MajorLink
- linkedChannel – The linked channel number.

```
static inline void EDMA__TcdSetBandWidth(edma_tcd_t *tcd, edma_bandwidth_t bandWidth)
```

Sets the bandwidth for the eDMA TCD.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- `tcd` – A pointer to the TCD structure.
- `bandWidth` – A bandwidth setting, which can be one of the following:
 - `kEDMABandwidthStallNone`
 - `kEDMABandwidthStall4Cycle`
 - `kEDMABandwidthStall8Cycle`

```
void EDMA__TcdSetModulo(edma_tcd_t *tcd, edma_modulo_t srcModulo, edma_modulo_t destModulo)
```

Sets the source modulo and the destination modulo for the eDMA TCD.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- `tcd` – A pointer to the TCD structure.
- `srcModulo` – A source modulo value.
- `destModulo` – A destination modulo value.

```
static inline void EDMA__TcdEnableAutoStopRequest(edma_tcd_t *tcd, bool enable)
```

Sets the auto stop request for the eDMA TCD.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- `tcd` – A pointer to the TCD structure.
- `enable` – The command to enable (true) or disable (false).

```
void EDMA__TcdEnableInterrupts(edma_tcd_t *tcd, uint32_t mask)
```

Enables the interrupt source for the eDMA TCD.

Parameters

- `tcd` – Point to the TCD structure.
- `mask` – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

```
void EDMA__TcdDisableInterrupts(edma_tcd_t *tcd, uint32_t mask)
```

Disables the interrupt source for the eDMA TCD.

Parameters

- `tcd` – Point to the TCD structure.
- `mask` – The mask of interrupt source to be set. Users need to use the defined `edma_interrupt_enable_t` type.

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

Enables the eDMA hardware channel request.

This function enables the hardware channel request.

Parameters

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

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

Disables the eDMA hardware channel request.

This function disables the hardware channel request.

Parameters

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

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

Starts the eDMA transfer by using the software trigger.

This function starts a minor loop transfer.

Parameters

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

```
uint32_t EDMA_GetRemainingMajorLoopCount(DMA_Type *base, uint32_t channel)
```

Gets the Remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.

- a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: $\text{RemainingBytes} = \text{RemainingMajorLoopCount} * \text{NBYTES}(\text{initially configured})$
-

Parameters

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

Returns

Major loop count which has not been transferred yet for the current TCD.

```
static inline uint32_t EDMA_GetErrorStatusFlags(DMA_Type *base)
```

Gets the eDMA channel error status flags.

Parameters

- base – eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the `_edma_error_status_flags` type to decode the return variables.

```
uint32_t EDMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)
```

Gets the eDMA channel status flags.

Parameters

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

Returns

The mask of channel status flags. Users need to use the `_edma_channel_status_flags` type to decode the return variables.

`void EDMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)`

Clears the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of channel status to be cleared. Users need to use the defined `_edma_channel_status_flags` type.

`void EDMA_CreateHandle(edma_handle_t *handle, DMA_Type *base, uint32_t channel)`

Creates the eDMA handle.

This function is called if using the transactional API for eDMA. This function initializes the internal state of the eDMA handle.

Parameters

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

`void EDMA_InstallTCDMemory(edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)`

Installs the TCDs memory pool into the eDMA handle.

This function is called after the `EDMA_CreateHandle` to use scatter/gather feature.

Parameters

- handle – eDMA handle pointer.
- 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.

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

`void EDMA_PrepareTransferConfig(edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)`

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- `config` – The user configuration structure of type `edma_transfer_config_t`.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `srcOffset` – eDMA transfer source address offset
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `destOffset` – eDMA transfer destination address offset
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.

```
void EDMA__PrepareTransfer(edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth,  
                           void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest,  
                           uint32_t transferBytes, edma_transfer_type_t transferType)
```

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- `config` – The user configuration structure of type `edma_transfer_config_t`.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.
- `transferType` – eDMA transfer type.

```
status_t EDMA__SubmitTransfer(edma_handle_t *handle, const edma_transfer_config_t *config)
```

Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- `handle` – eDMA handle pointer.
- `config` – Pointer to eDMA transfer configuration structure.

Return values

- `kStatus_EDMA_Success` – It means submit transfer request succeed.
- `kStatus_EDMA_QueueFull` – It means TCD queue is full. Submit transfer request is not allowed.
- `kStatus_EDMA_Busy` – It means the given channel is busy, need to submit request later.

`void EDMA_StartTransfer(edma_handle_t *handle)`

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- `handle` – eDMA handle pointer.

`void EDMA_StopTransfer(edma_handle_t *handle)`

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call `EDMA_StartTransfer()` again to resume the transfer.

Parameters

- `handle` – eDMA handle pointer.

`void EDMA_AbortTransfer(edma_handle_t *handle)`

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- `handle` – DMA handle pointer.

`static inline uint32_t EDMA_GetUnusedTCDNumber(edma_handle_t *handle)`

Get unused TCD slot number.

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

- `handle` – DMA handle pointer.

Returns

The unused tcd slot number.

`static inline uint32_t EDMA_GetNextTCDAddress(edma_handle_t *handle)`

Get the next tcd address.

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

- `handle` – DMA handle pointer.

Returns

The next TCD address.

`static inline edma_transfer_size_t EDMA_GetTransferSize(uint32_t width)`

Get the transfer size.

This function gets the transfer size.

Parameters

- `width` – transfer width(bytes).

Returns

The transfer size.

void EDMA_HandleIRQ(*edma_handle_t* *handle)

eDMA IRQ handler for the current major loop transfer completion.

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

Parameters

- handle – eDMA handle pointer.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.3.0.

Version 2.5.0.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.5.0.

enum _edma_transfer_size

eDMA transfer configuration

Values:

enumerator kEDMA_TransferSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kEDMA_TransferSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kEDMA_TransferSize4Bytes

Source/Destination data transfer size is 4 bytes every time

enumerator kEDMA_TransferSize8Bytes

Source/Destination data transfer size is 8 bytes every time

enumerator kEDMA_TransferSize16Bytes

Source/Destination data transfer size is 16 bytes every time

enumerator kEDMA_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enumerator kEDMA_TransferSize64Bytes

Source/Destination data transfer size is 64 bytes every time

enum _edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes
Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes
Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes
Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes
Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes
Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes
Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes
Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes
Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes
Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes
Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes
Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes
Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes
Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes
Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes
Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes
Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes
Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes
Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes
Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes
Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes
Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes

Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes

Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes

Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes

Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes

Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes

Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes

Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes

Circular buffer size is 2 G bytes.

enum _edma_bandwidth

Bandwidth control.

Values:

enumerator kEDMA_BandwidthStallNone

No eDMA engine stalls.

enumerator kEDMA_BandwidthStall4Cycle

eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_BandwidthStall8Cycle

eDMA engine stalls for 8 cycles after each read/write.

enum _edma_channel_link_type

Channel link type.

Values:

enumerator kEDMA_LinkNone

No channel link

enumerator kEDMA_MinorLink

Channel link after each minor loop

enumerator kEDMA_MajorLink

Channel link while major loop count exhausted

eDMA channel status flags, _edma_channel_status_flags

Values:

enumerator kEDMA_DoneFlag

DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag

eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag

eDMA interrupt flag, set while an interrupt occurred of this channel

eDMA channel error status flags, _edma_error_status_flags

Values:

enumerator kEDMA_DestinationBusErrorFlag

Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag

Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag

Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag

NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag

Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag

Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag

Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag

Source address not aligned with source size

enumerator kEDMA_TransferCanceledFlag

Transfer cancelled

enumerator kEDMA_ErrorChannelFlag

Error channel number of the cancelled channel number

enumerator kEDMA_ValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

eDMA channel system bus information, _edma_channel_sys_bus_info

Values:

enumerator kEDMA_PrivilegedAccessLevel

Privileged Access Level for DMA transfers. 0b - User protection level; 1b - Privileged protection level.

enumerator kEDMA_MasterId

DMA's master ID when channel is active and master ID replication is enabled.

enum _edma_interrupt_enable

eDMA interrupt source

Values:

enumerator kEDMA_ErrorInterruptEnable

Enable interrupt while channel error occurs.

enumerator kEDMA_MajorInterruptEnable

Enable interrupt while major count exhausted.

enumerator kEDMA_HalfInterruptEnable

Enable interrupt while major count to half value.

enum _edma_transfer_type

eDMA transfer type

Values:

enumerator kEDMA_MemoryToMemory

Transfer from memory to memory

enumerator kEDMA_PeripheralToMemory

Transfer from peripheral to memory

enumerator kEDMA_MemoryToPeripheral

Transfer from memory to peripheral

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.

enum _edma_transfer_size

eDMA transfer configuration

Values:

enumerator kEDMA_TransferSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kEDMA_TransferSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kEDMA_TransferSize4Bytes

Source/Destination data transfer size is 4 bytes every time

enumerator kEDMA_TransferSize8Bytes

Source/Destination data transfer size is 8 bytes every time

enumerator kEDMA_TransferSize16Bytes

Source/Destination data transfer size is 16 bytes every time

enumerator kEDMA_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enumerator kEDMA_TransferSize64Bytes

Source/Destination data transfer size is 64 bytes every time

enum _edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes
Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes
Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes
Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes
Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes
Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes
Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes
Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes
Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes
Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes
Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes
Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes
Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes
Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes
Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes
Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes
Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes
Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes
Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes
Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes
Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes
Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes

Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes

Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes

Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes

Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes

Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes

Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes

Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes

Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes

Circular buffer size is 2 G bytes.

enum _edma_bandwidth

Bandwidth control.

Values:

enumerator kEDMA_BandwidthStallNone

No eDMA engine stalls.

enumerator kEDMA_BandwidthStall4Cycle

eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_BandwidthStall8Cycle

eDMA engine stalls for 8 cycles after each read/write.

enum _edma_channel_link_type

Channel link type.

Values:

enumerator kEDMA_LinkNone

No channel link

enumerator kEDMA_MinorLink

Channel link after each minor loop

enumerator kEDMA_MajorLink

Channel link while major loop count exhausted

eDMA channel status flags, _edma_channel_status_flags

Values:

enumerator kEDMA_DoneFlag

DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag

eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag

eDMA interrupt flag, set while an interrupt occurred of this channel

eDMA channel error status flags, _edma_error_status_flags

Values:

enumerator kEDMA_DestinationBusErrorFlag

Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag

Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag

Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag

NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag

Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag

Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag

Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag

Source address not aligned with source size

enumerator kEDMA_TransferCanceledFlag

Transfer cancelled

enumerator kEDMA_ErrorChannelFlag

Error channel number of the cancelled channel number

enumerator kEDMA_ValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

eDMA channel system bus information, _edma_channel_sys_bus_info

Values:

enumerator kEDMA_PrivilegedAccessLevel

Privileged Access Level for DMA transfers. 0b - User protection level; 1b - Privileged protection level.

enumerator kEDMA_MasterId

DMA's master ID when channel is active and master ID replication is enabled.

enum _edma_interrupt_enable

eDMA interrupt source

Values:

enumerator kEDMA_ErrorInterruptEnable

Enable interrupt while channel error occurs.

enumerator kEDMA_MajorInterruptEnable
Enable interrupt while major count exhausted.

enumerator kEDMA_HalfInterruptEnable
Enable interrupt while major count to half value.

enum _edma_transfer_type
eDMA transfer type

Values:

enumerator kEDMA_MemoryToMemory
Transfer from memory to memory

enumerator kEDMA_PeripheralToMemory
Transfer from peripheral to memory

enumerator kEDMA_MemoryToPeripheral
Transfer from memory to peripheral

enumerator kEDMA_PeripheralToPeripheral
Transfer from Peripheral to peripheral

eDMA transfer status, _edma_transfer_status

Values:

enumerator kStatus_EDMA_QueueFull
TCD queue is full.

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

typedef enum _edma_transfer_size edma_transfer_size_t
eDMA transfer configuration

typedef enum _edma_modulo edma_modulo_t
eDMA modulo configuration

typedef enum _edma_bandwidth edma_bandwidth_t
Bandwidth control.

typedef enum _edma_channel_link_type edma_channel_link_type_t
Channel link type.

typedef enum _edma_interrupt_enable edma_interrupt_enable_t
eDMA interrupt source

typedef enum _edma_transfer_type edma_transfer_type_t
eDMA transfer type

typedef struct _edma_config edma_config_t
eDMA global configuration structure.

typedef struct _edma_transfer_config edma_transfer_config_t
eDMA transfer configuration
This structure configures the source/destination transfer attribute.

typedef struct _edma_channel_Preemption_config edma_channel_Preemption_config_t
eDMA channel priority configuration

```
typedef struct _edma_minor_offset_config edma__minor__offset__config__t
    eDMA minor offset configuration
```

```
typedef struct _edma_tcd edma__tcd__t
    eDMA TCD.
```

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

```
typedef void (*edma__callback)(struct _edma_handle *handle, void *userData, bool transferDone,
uint32_t tcDs)
```

Define callback function for eDMA.

```
typedef uint32_t (*edma__memorymap__callback)(uint32_t addr)
    Memroy map function callback for DMA.
```

```
typedef struct _edma_handle edma__handle__t
    eDMA transfer handle structure
```

```
typedef enum _edma_transfer_size edma__transfer__size__t
    eDMA transfer configuration
```

```
typedef enum _edma_modulo edma__modulo__t
    eDMA modulo configuration
```

```
typedef enum _edma_bandwidth edma__bandwidth__t
    Bandwidth control.
```

```
typedef enum _edma_channel_link_type edma__channel__link__type__t
    Channel link type.
```

```
typedef enum _edma_interrupt_enable edma__interrupt__enable__t
    eDMA interrupt source
```

```
typedef enum _edma_transfer_type edma__transfer__type__t
    eDMA transfer type
```

```
typedef struct _edma_config edma__config__t
    eDMA global configuration structure.
```

```
typedef struct _edma_transfer_config edma__transfer__config__t
    eDMA transfer configuration
```

This structure configures the source/destination transfer attribute.

```
typedef struct _edma_channel_Preemption_config edma__channel__Preemption__config__t
    eDMA channel priority configuration
```

```
typedef struct _edma_minor_offset_config edma__minor__offset__config__t
    eDMA minor offset configuration
```

```
typedef struct _edma_tcd edma__tcd__t
    eDMA TCD.
```

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

```
typedef void (*edma__callback)(struct _edma_handle *handle, void *userData, bool transferDone,
uint32_t tcDs)
```

Define callback function for eDMA.

```
typedef uint32_t (*edma__memorymap__callback)(uint32_t addr)
    Memroy map function callback for DMA.
```

```
typedef struct _edma_handle edma_handle_t  
    eDMA transfer handle structure
```

```
struct _edma_config  
    #include <fsl_ad_edma.h> eDMA global configuration structure.
```

Public Members

bool enableMasterIdReplication
 Enable (true) master ID replication. If Master ID replication is disabled, the privileged protection level (supervisor mode) for DMA transfers is used.

bool enableHaltOnError
 Enable (true) transfer halt on error. Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

bool enableRoundRobinArbitration
 Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection

bool enableDebugMode
 Enable(true) eDMA debug mode. When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

```
struct _edma_transfer_config  
    #include <fsl_ad_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_ad_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_ad_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_ad_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 stdc 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_ad_edma.h> eDMA transfer handle structure

Public Members

edma_callback callback
Callback function for major count exhausted.

void *userData
Callback function parameter.

DMA_AD_Type *base
eDMA peripheral base address.

edma_tcd_t *tcdPool
Pointer to memory stored TCDs.

uint8_t channel
eDMA channel number.

volatile int8_t header
The first TCD index.

volatile int8_t tail
The last TCD index.

volatile int8_t tcdUsed
The number of used TCD slots.

volatile int8_t tcdSize
The total number of TCD slots in the queue.

uint8_t flags
The status of the current channel.

DMA_Type *base
eDMA peripheral base address.

2.12 ENET: Ethernet MAC Driver

void ENET_GetDefaultConfig(*enet_config_t* *config)
Gets the ENET default configuration structure.

The purpose of this API is to get the default ENET MAC controller configure structure for ENET_Init(). User may use the initialized structure unchanged in ENET_Init(), or modify some fields of the structure before calling ENET_Init(). Example:

```
enet_config_t config;  
ENET_GetDefaultConfig(&config);
```

Parameters

- config – The ENET mac controller configuration structure pointer.

status_t ENET_Up(ENET_Type *base, *enet_handle_t* *handle, const *enet_config_t* *config, const *enet_buffer_config_t* *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)

Initializes the ENET module.

This function initializes the module with the ENET configuration.

Note: ENET has two buffer descriptors legacy buffer descriptors and enhanced IEEE 1588 buffer descriptors. The legacy descriptor is used by default. To use the IEEE 1588 feature, use the enhanced IEEE 1588 buffer descriptor by defining “ENET_ENHANCEDBUFFERDESCRIPTOR_MODE” and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Up().

Parameters

- base – ENET peripheral base address.
- handle – ENET handler pointer.
- config – ENET mac configuration structure pointer. The “enet_config_t” type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
- bufferConfig – ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization. It is the start address of “ringNum” enet_buffer_config structures. To support added multi-ring features in some soc and compatible with the previous enet driver version. For single ring supported, this bufferConfig is a buffer configure structure pointer, for multi-ring supported and used case, this bufferConfig pointer should be a buffer configure structure array pointer.
- macAddr – ENET mac address of Ethernet device. This MAC address should be provided.
- srcClock_Hz – The internal module clock source for MII clock.

Return values

- kStatus_Success – Succeed to initialize the ethernet driver.
- kStatus_ENET_InitMemoryFail – Init fails since buffer memory is not enough.

status_t ENET_Init(ENET_Type *base, *enet_handle_t* *handle, const *enet_config_t* *config, const *enet_buffer_config_t* *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)

Initializes the ENET module.

This function ungates the module clock and initializes it with the ENET configuration.

Note: ENET has two buffer descriptors legacy buffer descriptors and enhanced IEEE 1588 buffer descriptors. The legacy descriptor is used by default. To use the IEEE 1588 feature, use the enhanced IEEE 1588 buffer descriptor by defining “ENET_ENHANCEDBUFFERDESCRIPTOR_MODE” and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Init().

Parameters

- base – ENET peripheral base address.
- handle – ENET handler pointer.

- `config` – ENET mac configuration structure pointer. The “enet_config_t” type mac configuration return from `ENET_GetDefaultConfig` can be used directly. It is also possible to verify the Mac configuration using other methods.
- `bufferConfig` – ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization. It is the start address of “ringNum” `enet_buffer_config` structures. To support added multi-ring features in some soc and compatible with the previous enet driver version. For single ring supported, this `bufferConfig` is a buffer configure structure pointer, for multi-ring supported and used case, this `bufferConfig` pointer should be a buffer configure structure array pointer.
- `macAddr` – ENET mac address of Ethernet device. This MAC address should be provided.
- `srcClock_Hz` – The internal module clock source for MII clock.

Return values

- `kStatus_Success` – Succeed to initialize the ethernet driver.
- `kStatus_ENET_InitMemoryFail` – Init fails since buffer memory is not enough.

`void ENET_Down(ENET_Type *base)`

Stops the ENET module.

This function disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`void ENET_Deinit(ENET_Type *base)`

Deinitializes the ENET module.

This function gates the module clock, clears ENET interrupts, and disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`static inline void ENET_Reset(ENET_Type *base)`

Resets the ENET module.

This function restores the ENET module to reset state. Note that this function sets all registers to reset state. As a result, the ENET module can't work after calling this function.

Parameters

- `base` – ENET peripheral base address.

`void ENET_ResetHardware(void)`

Resets the ENET hardware.

This function resets ENET related resources in the hardware.

`void ENET_SetMII(ENET_Type *base, enet_mii_speed_t speed, enet_mii_duplex_t duplex)`

Sets the ENET MII speed and duplex.

This API is provided to dynamically change the speed and duplex for MAC.

Parameters

- `base` – ENET peripheral base address.
- `speed` – The speed of the RMII mode.

- duplex – The duplex of the RMII mode.

void ENET_SetSMI(ENET_Type *base, uint32_t srcClock_Hz, bool isPreambleDisabled)

Sets the ENET SMI(serial management interface)- MII management interface.

Parameters

- base – ENET peripheral base address.
- srcClock_Hz – This is the ENET module clock frequency. See clock distribution.
- isPreambleDisabled – The preamble disable flag.
 - true Enables the preamble.
 - false Disables the preamble.

static inline bool ENET_GetSMI(ENET_Type *base)

Gets the ENET SMI- MII management interface configuration.

This API is used to get the SMI configuration to check whether the MII management interface has been set.

Parameters

- base – ENET peripheral base address.

Returns

The SMI setup status true or false.

static inline uint32_t ENET_ReadSMIData(ENET_Type *base)

Reads data from the PHY register through an SMI interface.

Parameters

- base – ENET peripheral base address.

Returns

The data read from PHY

static inline void ENET_StartSMIWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, *enet_mii_write_t* operation, uint16_t data)

Sends the MDIO IEEE802.3 Clause 22 format write command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOWrite() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address. Range from 0 ~ 31.
- regAddr – The PHY register address. Range from 0 ~ 31.
- operation – The write operation.
- data – The data written to PHY.

static inline void ENET_StartSMIRead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, *enet_mii_read_t* operation)

Sends the MDIO IEEE802.3 Clause 22 format read command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIORead() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address. Range from 0 ~ 31.
- regAddr – The PHY register address. Range from 0 ~ 31.
- operation – The read operation.

status_t ENET_MDIOWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t data)
MDIO write with IEEE802.3 Clause 22 format.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address. Range from 0 ~ 31.
- regAddr – The PHY register. Range from 0 ~ 31.
- data – The data written to PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

status_t ENET_MDIORead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t *pData)

MDIO read with IEEE802.3 Clause 22 format.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address. Range from 0 ~ 31.
- regAddr – The PHY register. Range from 0 ~ 31.
- pData – The data read from PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

static inline void ENET_StartExtC45SMIWriteReg(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t regAddr)

Sends the MDIO IEEE802.3 Clause 45 format write register command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOC45Write()/ENET_MDIOC45Read() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.

static inline void ENET_StartExtC45SMIWriteData(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t data)

Sends the MDIO IEEE802.3 Clause 45 format write data command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOC45Write() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- data – The data written to PHY.

```
static inline void ENET_StartExtC45SMIReadData(ENET_Type *base, uint8_t portAddr, uint8_t devAddr)
```

Sends the MDIO IEEE802.3 Clause 45 format read data command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOC45Read() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.

```
status_t ENET_MDIOC45Write(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t regAddr, uint16_t data)
```

MDIO write with IEEE802.3 Clause 45 format.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.
- data – The data written to PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
status_t ENET_MDIOC45Read(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t regAddr, uint16_t *pData)
```

MDIO read with IEEE802.3 Clause 45 format.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.
- pData – The data read from PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
static inline void ENET_SetRGMIIClockDelay(ENET_Type *base, bool txEnabled, bool rxEnabled)
```

Control the usage of the delayed tx/rx RGMII clock.

Parameters

- base – ENET peripheral base address.
- txEnabled – Enable or disable to generate the delayed version of RGMII_TXC.
- rxEnabled – Enable or disable to use the delayed version of RGMII_RXC.

```
void ENET_SetMacAddr(ENET_Type *base, uint8_t *macAddr)
```

Sets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.
- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

```
void ENET_GetMacAddr(ENET_Type *base, uint8_t *macAddr)
```

Gets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.
- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

```
void ENET_AddMulticastGroup(ENET_Type *base, uint8_t *address)
```

Adds the ENET device to a multicast group.

Parameters

- base – ENET peripheral base address.
- address – The six-byte multicast group address which is provided by application.

```
void ENET_LeaveMulticastGroup(ENET_Type *base, uint8_t *address)
```

Moves the ENET device from a multicast group.

Parameters

- base – ENET peripheral base address.
- address – The six-byte multicast group address which is provided by application.

```
static inline void ENET_ActiveRead(ENET_Type *base)
```

Activates frame reception for multiple rings.

This function is to active the enet read process.

Note: This must be called after the MAC configuration and state are ready. It must be called after the ENET_Init(). This should be called when the frame reception is required.

Parameters

- base – ENET peripheral base address.

```
static inline void ENET_EnableSleepMode(ENET_Type *base, bool enable)
```

Enables/disables the MAC to enter sleep mode. This function is used to set the MAC enter sleep mode. When entering sleep mode, the magic frame wakeup interrupt should be enabled to wake up MAC from the sleep mode and reset it to normal mode.

Parameters

- base – ENET peripheral base address.
- enable – True enable sleep mode, false disable sleep mode.

```
static inline void ENET_GetAccelFunction(ENET_Type *base, uint32_t *txAccelOption, uint32_t *rxAccelOption)
```

Gets ENET transmit and receive accelerator functions from MAC controller.

Parameters

- base – ENET peripheral base address.
- txAccelOption – The transmit accelerator option. The “enet_tx_accelerator_t” is recommended to be used to as the mask to get the exact the accelerator option.
- rxAccelOption – The receive accelerator option. The “enet_rx_accelerator_t” is recommended to be used to as the mask to get the exact the accelerator option.

```
static inline void ENET_EnableInterrupts(ENET_Type *base, uint32_t mask)
```

Enables the ENET interrupt.

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enumeration members. See `enet_interrupt_enable_t`. For example, to enable the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_EnableInterrupts(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- base – ENET peripheral base address.
- mask – ENET interrupts to enable. This is a logical OR of the enumeration `enet_interrupt_enable_t`.

```
static inline void ENET_DisableInterrupts(ENET_Type *base, uint32_t mask)
```

Disables the ENET interrupt.

This function disables the ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `enet_interrupt_enable_t`. For example, to disable the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_DisableInterrupts(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- base – ENET peripheral base address.
- mask – ENET interrupts to disable. This is a logical OR of the enumeration `enet_interrupt_enable_t`.

```
static inline uint32_t ENET_GetInterruptStatus(ENET_Type *base)
```

Gets the ENET interrupt status flag.

Parameters

- base – ENET peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration `enet_interrupt_enable_t`.

```
static inline void ENET_ClearInterruptStatus(ENET_Type *base, uint32_t mask)
```

Clears the ENET interrupt events status flag.

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the `enet_interrupt_enable_t`. For example, to clear the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_ClearInterruptStatus(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- base – ENET peripheral base address.
- mask – ENET interrupt source to be cleared. This is the logical OR of members of the enumeration `enet_interrupt_enable_t`.

```
void ENET_SetRxISRHandler(ENET_Type *base, enet_isr_t ISRHandler)
```

Set the second level Rx IRQ handler.

Parameters

- base – ENET peripheral base address.
- ISRHandler – The handler to install.

```
void ENET_SetTxISRHandler(ENET_Type *base, enet_isr_t ISRHandler)
```

Set the second level Tx IRQ handler.

Parameters

- base – ENET peripheral base address.
- ISRHandler – The handler to install.

```
void ENET_SetErrISRHandler(ENET_Type *base, enet_isr_t ISRHandler)
```

Set the second level Err IRQ handler.

Parameters

- base – ENET peripheral base address.
- ISRHandler – The handler to install.

```
void ENET_GetRxErrBeforeReadFrame(enet_handle_t *handle, enet_data_error_stats_t  
                                *eErrorStatic, uint8_t ringId)
```

Gets the error statistics of a received frame for ENET specified ring.

This API must be called after the `ENET_GetRxFrameSize` and before the `ENET_ReadFrame()`. If the `ENET_GetRxFrameSize` returns `kStatus_ENET_RxFrameError`, the `ENET_GetRxErrBeforeReadFrame` can be used to get the exact error statistics. This is an example.

```
status = ENET_GetRxFrameSize(&g_handle, &length, 0);  
if (status == kStatus_ENET_RxFrameError)  
{  
    Comments: Get the error information of the received frame.
```

(continues on next page)

(continued from previous page)

```

ENET_GetRxErrBeforeReadFrame(&g_handle, &eErrStatic, 0);
Comments: update the receive buffer.
ENET_ReadFrame(EXAMPLE_ENET, &g_handle, NULL, 0);
}

```

Parameters

- handle – The ENET handler structure pointer. This is the same handler pointer used in the ENET_Init.
- eErrorStatic – The error statistics structure pointer.
- ringId – The ring index, range from 0 ~ (FSL_FEATURE_ENET_INSTANCE_QUEUE(x) - 1).

void ENET_EnableStatistics(ENET_Type *base, bool enable)

Enables/disables collection of transfer statistics.

Note that this function does not reset any of the already collected data, use the function ENET_ResetStatistics to clear the transfer statistics if needed.

Parameters

- base – ENET peripheral base address.
- enable – True enable statistics collection, false disable statistics collection.

void ENET_GetStatistics(ENET_Type *base, *enet_transfer_stats_t* *statistics)

Gets transfer statistics.

Copies the actual value of hardware counters into the provided structure. Calling this function does not reset the counters in hardware.

Parameters

- base – ENET peripheral base address.
- statistics – The statistics structure pointer.

void ENET_ResetStatistics(ENET_Type *base)

Resets transfer statistics.

Sets the value of hardware transfer counters to zero.

Parameters

- base – ENET peripheral base address.

status_t ENET_GetRxFrameSize(*enet_handle_t* *handle, uint32_t *length, uint8_t ringId)

Gets the size of the read frame for specified ring.

This function gets a received frame size from the ENET buffer descriptors.

Note: The FCS of the frame is automatically removed by MAC and the size is the length without the FCS. After calling ENET_GetRxFrameSize, ENET_ReadFrame() should be called to receive frame and update the BD if the result is not “kStatus_ENET_RxFrameEmpty”.

Parameters

- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.
- length – The length of the valid frame received.
- ringId – The ring index or ring number.

Return values

- `kStatus_ENET_RxFrameEmpty` – No frame received. Should not call `ENET_ReadFrame` to read frame.
- `kStatus_ENET_RxFrameError` – Data error happens. `ENET_ReadFrame` should be called with NULL data and NULL length to update the receive buffers.
- `kStatus_Success` – Receive a frame Successfully then the `ENET_ReadFrame` should be called with the right data buffer and the captured data length input.

`status_t` `ENET_ReadFrame`(`ENET_Type` *base, `enet_handle_t` *handle, `uint8_t` *data, `uint32_t` length, `uint8_t` ringId, `uint32_t` *ts)

Reads a frame from the ENET device. This function reads a frame (both the data and the length) from the ENET buffer descriptors. User can get timestamp through ts pointer if the ts is not NULL.

Note: It doesn't store the timestamp in the receive timestamp queue. The `ENET_GetRxFrameSize` should be used to get the size of the prepared data buffer. This API uses memcpy to copy data from DMA buffer to application buffer, 4 bytes aligned data buffer in 32 bits platforms provided by user may let compiler use optimization instruction to reduce time consumption. This is an example:

```
uint32_t length;
enet_handle_t g_handle;
Comments: Get the received frame size firstly.
status = ENET_GetRxFrameSize(&g_handle, &length, 0);
if (length != 0)
{
    Comments: Allocate memory here with the size of "length"
    uint8_t *data = memory allocate interface;
    if (!data)
    {
        ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0, NULL);
        Comments: Add the console warning log.
    }
    else
    {
        status = ENET_ReadFrame(ENET, &g_handle, data, length, 0, NULL);
        Comments: Call stack input API to deliver the data to stack
    }
}
else if (status == kStatus_ENET_RxFrameError)
{
    Comments: Update the received buffer when a error frame is received.
    ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0, NULL);
}
```

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler structure. This is the same handler pointer used in the `ENET_Init`.
- data – The data buffer provided by user to store the frame which memory size should be at least "length".
- length – The size of the data buffer which is still the length of the received frame.

- ringId – The ring index or ring number.
- ts – The timestamp address to store received timestamp.

Returns

The execute status, successful or failure.

status_t ENET_SendFrame(ENET_Type *base, *enet_handle_t* *handle, const uint8_t *data, uint32_t length, uint8_t ringId, bool tsFlag, void *context)

Transmits an ENET frame for specified ring.

Note: The CRC is automatically appended to the data. Input the data to send without the CRC. This API uses memcpy to copy data from DMA buffer to application buffer, 4 bytes aligned data buffer in 32 bits platforms provided by user may let compiler use optimization instruction to reduce time consumption.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- data – The data buffer provided by user to send.
- length – The length of the data to send.
- ringId – The ring index or ring number.
- tsFlag – Timestamp enable flag.
- context – Used by user to handle some events after transmit over.

Return values

- kStatus_Success – Send frame succeed.
- kStatus_ENET_TxFrameBusy – Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added after each call return with kStatus_ENET_TxFrameBusy.

status_t ENET_SetTxReclaim(*enet_handle_t* *handle, bool isEnabled, uint8_t ringId)

Enable or disable tx descriptors reclaim mechanism.

Note: This function must be called when no pending send frame action. Set enable if you want to reclaim context or timestamp in interrupt.

Parameters

- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- isEnabled – Enable or disable flag.
- ringId – The ring index or ring number.

Return values

- kStatus_Success – Succeed to enable/disable Tx reclaim.
- kStatus_Fail – Fail to enable/disable Tx reclaim.

`void ENET_ReclaimTxDescriptor(ENET_Type *base, enet_handle_t *handle, uint8_t ringId)`

Reclaim tx descriptors. This function is used to update the tx descriptor status and store the tx timestamp when the 1588 feature is enabled. This is called by the transmit interrupt IRQ handler after the complete of a frame transmission.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- ringId – The ring index or ring number.

`status_t ENET_GetRxFrame(ENET_Type *base, enet_handle_t *handle, enet_rx_frame_struct_t *rxFrame, uint8_t ringId)`

Receives one frame in specified BD ring with zero copy.

This function uses the user-defined allocation and free callbacks. Every time application gets one frame through this function, driver stores the buffer address(es) in `enet_buffer_struct_t` and allocate new buffer(s) for the BD(s). If there's no memory buffer in the pool, this function drops current one frame to keep the Rx frame in BD ring is as fresh as possible.

Note: Application must provide a memory pool including at least BD number + n buffers in order for this function to work properly, because each BD must always take one buffer while driver is running, then other extra n buffer(s) can be taken by application. Here n is the `ceil(max_frame_length(set by RCR) / bd_rx_size(set by MRBR))`. Application must also provide an array structure in `rxFrame->rxBuffArray` with n index to receive one complete frame in any case.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- rxFrame – The received frame information structure provided by user.
- ringId – The ring index or ring number.

Return values

- `kStatus_Success` – Succeed to get one frame and allocate new memory for Rx buffer.
- `kStatus_ENET_RxFrameEmpty` – There's no Rx frame in the BD.
- `kStatus_ENET_RxFrameError` – There's issue in this receiving.
- `kStatus_ENET_RxFrameDrop` – There's no new buffer memory for BD, drop this frame.

`status_t ENET_StartTxFrame(ENET_Type *base, enet_handle_t *handle, enet_tx_frame_struct_t *txFrame, uint8_t ringId)`

Sends one frame in specified BD ring with zero copy.

This function supports scattered buffer transmit, user needs to provide the buffer array.

Note: Tx reclaim should be enabled to ensure the Tx buffer ownership can be given back to application after Tx is over.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- txFrame – The Tx frame structure.
- ringId – The ring index or ring number.

Return values

- kStatus_Success – Succeed to send one frame.
- kStatus_ENET_TxFrameBusy – The BD is not ready for Tx or the reclaim operation still not finishes.
- kStatus_ENET_TxFrameOverLen – The Tx frame length is over max ethernet frame length.

void ENET_TransmitIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

The transmit IRQ handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_ReceiveIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

The receive IRQ handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_ErrorIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

Some special IRQ handler including the error, mii, wakeup irq handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_Ptp1588IRQHandler(ENET_Type *base)

the common IRQ handler for the 1588 irq handler.

This is used for the 1588 timer interrupt.

Parameters

- base – ENET peripheral base address.

void ENET_CommonFrame0IRQHandler(ENET_Type *base)

the common IRQ handler for the tx/rx/error etc irq handler.

This is used for the combined tx/rx/error interrupt for single/mutli-ring (frame 0).

Parameters

- base – ENET peripheral base address.

FSL_ENET_DRIVER_VERSION

Defines the driver version.

ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK

Empty bit mask.

ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK

Software owner one mask.

ENET_BUFFDESCRIPTOR_RX_WRAP_MASK

Next buffer descriptor is the start address.

ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask

Software owner two mask.

ENET_BUFFDESCRIPTOR_RX_LAST_MASK

Last BD of the frame mask.

ENET_BUFFDESCRIPTOR_RX_MISS_MASK

Received because of the promiscuous mode.

ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK

Broadcast packet mask.

ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK

Multicast packet mask.

ENET_BUFFDESCRIPTOR_RX_LENVIOLATE_MASK

Length violation mask.

ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK

Non-octet aligned frame mask.

ENET_BUFFDESCRIPTOR_RX_CRC_MASK

CRC error mask.

ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK

FIFO overrun mask.

ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK

Frame is truncated mask.

ENET_BUFFDESCRIPTOR_TX_READY_MASK

Ready bit mask.

ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK

Software owner one mask.

ENET_BUFFDESCRIPTOR_TX_WRAP_MASK

Wrap buffer descriptor mask.

ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK

Software owner two mask.

ENET_BUFFDESCRIPTOR_TX_LAST_MASK

Last BD of the frame mask.

ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK

Transmit CRC mask.

ENET_FRAME_MAX_FRAMELEN

Default maximum Ethernet frame size without VLAN tag.

ENET_FRAME_VLAN_TAGLEN

Ethernet single VLAN tag size.

ENET_FRAME_CRC_LEN

CRC size in a frame.

ENET_FRAME_TX_LEN_LIMITATION(x)

ENET_FIFO_MIN_RX_FULL

ENET minimum receive FIFO full.

ENET_RX_MIN_BUFFERSIZE

ENET minimum buffer size.

ENET_PHY_MAXADDRESS

Maximum PHY address.

ENET_TX_INTERRUPT

Enet Tx interrupt flag.

ENET_RX_INTERRUPT

Enet Rx interrupt flag.

ENET_TS_INTERRUPT

Enet timestamp interrupt flag.

ENET_ERR_INTERRUPT

Enet error interrupt flag.

Defines the status return codes for transaction.

Values:

enumerator kStatus_ENET_InitMemoryFail

Init fails since buffer memory is not enough.

enumerator kStatus_ENET_RxFrameError

A frame received but data error happen.

enumerator kStatus_ENET_RxFrameFail

Failed to receive a frame.

enumerator kStatus_ENET_RxFrameEmpty

No frame arrive.

enumerator kStatus_ENET_RxFrameDrop

Rx frame is dropped since no buffer memory.

enumerator kStatus_ENET_TxFrameOverLen

Tx frame over length.

enumerator kStatus_ENET_TxFrameBusy

Tx buffer descriptors are under process.

enumerator kStatus_ENET_TxFrameFail

Transmit frame fail.

enum _enet_mii_mode

Defines the MII/RMII/RGMII mode for data interface between the MAC and the PHY.

Values:

enumerator kENET_MiiMode

MII mode for data interface.

enumerator kENET_RmiiMode

RMII mode for data interface.

enumerator kENET_RgmiiMode
RGMII mode for data interface.

enum _enet_mii_speed
Defines the 10/100/1000 Mbps speed for the MII data interface.
Notice: “kENET_MiiSpeed1000M” only supported when mii mode is “kENET_RgmiiMode”.
Values:

enumerator kENET_MiiSpeed10M
Speed 10 Mbps.

enumerator kENET_MiiSpeed100M
Speed 100 Mbps.

enumerator kENET_MiiSpeed1000M
Speed 1000M bps.

enum _enet_mii_duplex
Defines the half or full duplex for the MII data interface.
Values:

enumerator kENET_MiiHalfDuplex
Half duplex mode.

enumerator kENET_MiiFullDuplex
Full duplex mode.

enum _enet_mii_write
Define the MII opcode for normal MDIO_CLAUSES_22 Frame.
Values:

enumerator kENET_MiiWriteNoCompliant
Write frame operation, but not MII-compliant.

enumerator kENET_MiiWriteValidFrame
Write frame operation for a valid MII management frame.

enum _enet_mii_read
Defines the read operation for the MII management frame.
Values:

enumerator kENET_MiiReadValidFrame
Read frame operation for a valid MII management frame.

enumerator kENET_MiiReadNoCompliant
Read frame operation, but not MII-compliant.

enum _enet_mii_extend_opcode
Define the MII opcode for extended MDIO_CLAUSES_45 Frame.
Values:

enumerator kENET_MiiAddrWrite_C45
Address Write operation.

enumerator kENET_MiiWriteFrame_C45
Write frame operation for a valid MII management frame.

enumerator kENET_MiiReadFrame_C45
Read frame operation for a valid MII management frame.

enum _enet_special_control_flag

Defines a special configuration for ENET MAC controller.

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to mac-SpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifo-FullThreshold and txFifoWatermark in the enet_config_t.

Values:

enumerator kENET_ControlFlowControlEnable

Enable ENET flow control: pause frame.

enumerator kENET_ControlRxPayloadCheckEnable

Enable ENET receive payload length check.

enumerator kENET_ControlRxPadRemoveEnable

Padding is removed from received frames.

enumerator kENET_ControlRxBroadCastRejectEnable

Enable broadcast frame reject.

enumerator kENET_ControlMacAddrInsert

Enable MAC address insert.

enumerator kENET_ControlStoreAndFwdDisable

Enable FIFO store and forward.

enumerator kENET_ControlSMIPreambleDisable

Enable SMI preamble.

enumerator kENET_ControlPromiscuousEnable

Enable promiscuous mode.

enumerator kENET_ControlMIILoopEnable

Enable ENET MII loop back.

enumerator kENET_ControlVLANTagEnable

Enable normal VLAN (single vlan tag).

enumerator kENET_ControlSVLANEnable

Enable S-VLAN.

enumerator kENET_ControlVLANUseSecondTag

Enable extracting the second vlan tag for further processing.

enum _enet_interrupt_enable

List of interrupts supported by the peripheral. This enumeration uses one-bit encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Values:

enumerator kENET_BabrInterrupt

Babbling receive error interrupt source

enumerator kENET_BabtInterrupt

Babbling transmit error interrupt source

enumerator kENET_GraceStopInterrupt

Graceful stop complete interrupt source

enumerator kENET_TxFrameInterrupt
TX FRAME interrupt source

enumerator kENET_TxBufferInterrupt
TX BUFFER interrupt source

enumerator kENET_RxFrameInterrupt
RX FRAME interrupt source

enumerator kENET_RxBufferInterrupt
RX BUFFER interrupt source

enumerator kENET_MiiInterrupt
MII interrupt source

enumerator kENET_EBusERInterrupt
Ethernet bus error interrupt source

enumerator kENET_LateCollisionInterrupt
Late collision interrupt source

enumerator kENET_RetryLimitInterrupt
Collision Retry Limit interrupt source

enumerator kENET_UnderrunInterrupt
Transmit FIFO underrun interrupt source

enumerator kENET_PayloadRxInterrupt
Payload Receive error interrupt source

enumerator kENET_WakeupInterrupt
WAKEUP interrupt source

enumerator kENET_TsAvailInterrupt
TS AVAIL interrupt source for PTP

enumerator kENET_TsTimerInterrupt
TS WRAP interrupt source for PTP

enum _enet_event

Defines the common interrupt event for callback use.

Values:

enumerator kENET_RxEvent
Receive event.

enumerator kENET_TxEvent
Transmit event.

enumerator kENET_ErrEvent
Error event: BABR/BABT/EBERR/LC/RL/UN/PLR .

enumerator kENET_WakeUpEvent
Wake up from sleep mode event.

enumerator kENET_TimeStampEvent
Time stamp event.

enumerator kENET_TimeStampAvailEvent
Time stamp available event.

enum _enet_idle_slope

Defines certain idle slope for bandwidth fraction.

Values:

enumerator kENET_IdleSlope1

The bandwidth fraction is about 0.002.

enumerator kENET_IdleSlope2

The bandwidth fraction is about 0.003.

enumerator kENET_IdleSlope4

The bandwidth fraction is about 0.008.

enumerator kENET_IdleSlope8

The bandwidth fraction is about 0.02.

enumerator kENET_IdleSlope16

The bandwidth fraction is about 0.03.

enumerator kENET_IdleSlope32

The bandwidth fraction is about 0.06.

enumerator kENET_IdleSlope64

The bandwidth fraction is about 0.11.

enumerator kENET_IdleSlope128

The bandwidth fraction is about 0.20.

enumerator kENET_IdleSlope256

The bandwidth fraction is about 0.33.

enumerator kENET_IdleSlope384

The bandwidth fraction is about 0.43.

enumerator kENET_IdleSlope512

The bandwidth fraction is about 0.50.

enumerator kENET_IdleSlope640

The bandwidth fraction is about 0.56.

enumerator kENET_IdleSlope768

The bandwidth fraction is about 0.60.

enumerator kENET_IdleSlope896

The bandwidth fraction is about 0.64.

enumerator kENET_IdleSlope1024

The bandwidth fraction is about 0.67.

enumerator kENET_IdleSlope1152

The bandwidth fraction is about 0.69.

enumerator kENET_IdleSlope1280

The bandwidth fraction is about 0.71.

enumerator kENET_IdleSlope1408

The bandwidth fraction is about 0.73.

enumerator kENET_IdleSlope1536

The bandwidth fraction is about 0.75.

enum `_enet_tx_accelerator`

Defines the transmit accelerator configuration.

Note that the hardware does not insert ICMPv6 protocol checksums as mentioned in errata ERR052152.

Values:

enumerator `kENET_TxAccelIsShift16Enabled`

Transmit FIFO shift-16.

enumerator `kENET_TxAccelIpCheckEnabled`

Insert IP header checksum.

enumerator `kENET_TxAccelProtoCheckEnabled`

Insert protocol checksum (TCP, UDP, ICMPv4).

enum `_enet_rx_accelerator`

Defines the receive accelerator configuration.

Note that the hardware does not validate ICMPv6 protocol checksums as mentioned in errata ERR052152.

Values:

enumerator `kENET_RxAccelPadRemoveEnabled`

Padding removal for short IP frames.

enumerator `kENET_RxAccelIpCheckEnabled`

Discard with wrong IP header checksum.

enumerator `kENET_RxAccelProtoCheckEnabled`

Discard with wrong protocol checksum (TCP, UDP, ICMPv4).

enumerator `kENET_RxAccelMacCheckEnabled`

Discard with Mac layer errors.

enumerator `kENET_RxAccelIsShift16Enabled`

Receive FIFO shift-16.

typedef enum `_enet_mii_mode` `enet_mii_mode_t`

Defines the MII/RMII/RGMII mode for data interface between the MAC and the PHY.

typedef enum `_enet_mii_speed` `enet_mii_speed_t`

Defines the 10/100/1000 Mbps speed for the MII data interface.

Notice: “kENET_MiiSpeed1000M” only supported when mii mode is “kENET_RgmiiMode”.

typedef enum `_enet_mii_duplex` `enet_mii_duplex_t`

Defines the half or full duplex for the MII data interface.

typedef enum `_enet_mii_write` `enet_mii_write_t`

Define the MII opcode for normal MDIO_CLAUSES_22 Frame.

typedef enum `_enet_mii_read` `enet_mii_read_t`

Defines the read operation for the MII management frame.

typedef enum `_enet_mii_extend_opcode` `enet_mii_extend_opcode`

Define the MII opcode for extended MDIO_CLAUSES_45 Frame.

typedef enum *_enet_special_control_flag* enet_special_control_flag_t

Defines a special configuration for ENET MAC controller.

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to macSpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifoFullThreshold and txFifoWatermark in the enet_config_t.

typedef enum *_enet_interrupt_enable* enet_interrupt_enable_t

List of interrupts supported by the peripheral. This enumeration uses one-bit encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

typedef enum *_enet_event* enet_event_t

Defines the common interrupt event for callback use.

typedef enum *_enet_idle_slope* enet_idle_slope_t

Defines certain idle slope for bandwidth fraction.

typedef enum *_enet_tx_accelerator* enet_tx_accelerator_t

Defines the transmit accelerator configuration.

Note that the hardware does not insert ICMPv6 protocol checksums as mentioned in errata ERR052152.

typedef enum *_enet_rx_accelerator* enet_rx_accelerator_t

Defines the receive accelerator configuration.

Note that the hardware does not validate ICMPv6 protocol checksums as mentioned in errata ERR052152.

typedef struct *_enet_rx_bd_struct* enet_rx_bd_struct_t

Defines the receive buffer descriptor structure for the little endian system.

typedef struct *_enet_tx_bd_struct* enet_tx_bd_struct_t

Defines the enhanced transmit buffer descriptor structure for the little endian system.

typedef struct *_enet_data_error_stats* enet_data_error_stats_t

Defines the ENET data error statistics structure.

typedef struct *_enet_rx_frame_error* enet_rx_frame_error_t

Defines the Rx frame error structure.

typedef struct *_enet_transfer_stats* enet_transfer_stats_t

Defines the ENET transfer statistics structure.

typedef struct *enet_frame_info* enet_frame_info_t

Defines the frame info structure.

typedef struct *_enet_tx_dirty_ring* enet_tx_dirty_ring_t

Defines the ENET transmit dirty addresses ring/queue structure.

typedef void (*enet_rx_alloc_callback_t)(ENET_Type *base, void *userData, uint8_t ringId)

Defines the ENET Rx memory buffer alloc function pointer.

typedef void (*enet_rx_free_callback_t)(ENET_Type *base, void *buffer, void *userData, uint8_t ringId)

Defines the ENET Rx memory buffer free function pointer.

```
typedef struct _enet_buffer_config enet_buffer_config_t
```

Defines the receive buffer descriptor configuration structure.

Note that for the internal DMA requirements, the buffers have a corresponding alignment requirements.

- a. The aligned receive and transmit buffer size must be evenly divisible by ENET_BUFF_ALIGNMENT. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.
- b. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by ENET_BUFF_ALIGNMENT. buffer descriptors should be put in non-cacheable region when cache is enabled.
- c. The aligned transmit and receive data buffer start address must be evenly divisible by ENET_BUFF_ALIGNMENT. Receive buffers should be continuous with the total size equal to "rxBdNumber * rxBuffSizeAlign". Transmit buffers should be continuous with the total size equal to "txBdNumber * txBuffSizeAlign". when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.

```
typedef struct _enet_intcoalesce_config enet_intcoalesce_config_t
```

Defines the interrupt coalescing configure structure.

```
typedef struct _enet_avb_config enet_avb_config_t
```

Defines the ENET AVB Configure structure.

This is used for to configure the extended ring 1 and ring 2.

- a. The classification match format is $(CMP3 \ll 12) \mid (CMP2 \ll 8) \mid (CMP1 \ll 4) \mid CMP0$. composed of four 3-bit compared VLAN priority field cmp0~cmp3, cm0 ~ cmp3 are used in parallel.

If CMP1,2,3 are not unused, please set them to the same value as CMP0.

- a. The idleSlope is used to calculate the Band Width fraction, BW fraction = $1 / (1 + 512/idleSlope)$. For avb configuration, the BW fraction of Class 1 and Class 2 combined must not exceed 0.75.

```
typedef struct _enet_handle enet_handle_t
```

```
typedef void (*enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, enet_frame_info_t *frameInfo, void *userData)
```

ENET callback function.

```
typedef struct _enet_config enet_config_t
```

Defines the basic configuration structure for the ENET device.

Note:

- a. macSpecialConfig is used for a special control configuration, A logical OR of "enet_special_control_flag_t". For a special configuration for MAC, set this parameter to 0.
- b. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO 3 - 192 bytes written to TX FIFO The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
- c. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULL and the maximum is

0xFF. If the end of the frame is stored in FIFO and the frame size is smaller than the txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.

- d. When “kENET_ControlFlowControlEnable” is set in the macSpecialConfig, ensure that the pauseDuration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.
- e. When “kENET_ControlStoreAndFwdDisabled” is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
- f. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The “enet_tx_accelerator_t” and “enet_rx_accelerator_t” are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.
- g. The intCoalesceCfg can be used in the rx or tx enabled cases to decrease the CPU loading.

```
typedef struct _enet_tx_bd_ring enet_tx_bd_ring_t
```

Defines the ENET transmit buffer descriptor ring/queue structure.

```
typedef struct _enet_rx_bd_ring enet_rx_bd_ring_t
```

Defines the ENET receive buffer descriptor ring/queue structure.

```
typedef struct _enet_buffer_struct enet_buffer_struct_t
```

```
typedef struct _enet_rx_frame_attribute_struct enet_rx_frame_attribute_t
```

```
typedef struct _enet_rx_frame_struct enet_rx_frame_struct_t
```

```
typedef struct _enet_tx_frame_struct enet_tx_frame_struct_t
```

```
typedef void (*enet_isr_t)(ENET_Type *base, enet_handle_t *handle)
```

Define interrupt IRQ handler.

```
const clock_ip_name_t s_enetClock[]
```

Pointers to enet clocks for each instance.

```
const clock_ip_name_t s_enetExtraClock[]
```

```
uint32_t ENET_GetInstance(ENET_Type *base)
```

Get the ENET instance from peripheral base address.

Parameters

- base – ENET peripheral base address.

Returns

ENET instance.

```
ENET_BUFFDESCRIPTOR_RX_ERR_MASK
```

Defines the receive error status flag mask.

```
struct _enet_rx_bd_struct
```

#include <fsl_enet.h> Defines the receive buffer descriptor structure for the little endian system.

Public Members

```
uint16_t length
```

Buffer descriptor data length.

uint16_t control

Buffer descriptor control and status.

uint32_t buffer

Data buffer pointer.

struct __enet_tx_bd_struct

#include <fsl_enet.h> Defines the enhanced transmit buffer descriptor structure for the little endian system.

Public Members

uint16_t length

Buffer descriptor data length.

uint16_t control

Buffer descriptor control and status.

uint32_t buffer

Data buffer pointer.

struct __enet_data_error_stats

#include <fsl_enet.h> Defines the ENET data error statistics structure.

Public Members

uint32_t statsRxLenGreaterErr

Receive length greater than RCR[MAX_FL].

uint32_t statsRxAlignErr

Receive non-octet alignment/

uint32_t statsRxFcsErr

Receive CRC error.

uint32_t statsRxOverRunErr

Receive over run.

uint32_t statsRxTruncateErr

Receive truncate.

struct __enet_rx_frame_error

#include <fsl_enet.h> Defines the Rx frame error structure.

Public Members

bool statsRxTruncateErr

Receive truncate.

bool statsRxOverRunErr

Receive over run.

bool statsRxFcsErr

Receive CRC error.

bool statsRxAlignErr

Receive non-octet alignment.

bool statsRxLenGreaterErr
Receive length greater than RCR[MAX_FL].

struct __enet__transfer_stats
#include <fsl_enet.h> Defines the ENET transfer statistics structure.

Public Members

uint32_t statsRxFrameCount
Rx frame number.

uint32_t statsRxFrameOk
Good Rx frame number.

uint32_t statsRxCrcErr
Rx frame number with CRC error.

uint32_t statsRxAlignErr
Rx frame number with alignment error.

uint32_t statsRxDropInvalidSFD
Dropped frame number due to invalid SFD.

uint32_t statsRxFifoOverflowErr
Rx FIFO overflow count.

uint32_t statsTxFrameCount
Tx frame number.

uint32_t statsTxFrameOk
Good Tx frame number.

uint32_t statsTxCrcAlignErr
The transmit frame is error.

uint32_t statsTxFifoUnderRunErr
Tx FIFO underrun count.

struct enet__frame__info
#include <fsl_enet.h> Defines the frame info structure.

Public Members

void *context
User specified data

struct __enet__tx__dirty__ring
#include <fsl_enet.h> Defines the ENET transmit dirty addresses ring/queue structure.

Public Members

enet_frame_info_t *txDirtyBase
Dirty buffer descriptor base address pointer.

uint16_t txGenIdx
tx generate index.

uint16_t txConsumeIdx
tx consume index.

uint16_t txRingLen

tx ring length.

bool isFull

tx ring is full flag.

struct __enet_buffer_config

#include <fsl_enet.h> Defines the receive buffer descriptor configuration structure.

Note that for the internal DMA requirements, the buffers have a corresponding alignment requirements.

- a. The aligned receive and transmit buffer size must be evenly divisible by ENET_BUFF_ALIGNMENT. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of “ENET_BUFF_ALIGNMENT” and the cache line size.
- b. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by ENET_BUFF_ALIGNMENT. buffer descriptors should be put in non-cacheable region when cache is enabled.
- c. The aligned transmit and receive data buffer start address must be evenly divisible by ENET_BUFF_ALIGNMENT. Receive buffers should be continuous with the total size equal to “rxBdNumber * rxBuffSizeAlign”. Transmit buffers should be continuous with the total size equal to “txBdNumber * txBuffSizeAlign”. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of “ENET_BUFF_ALIGNMENT” and the cache line size.

Public Members

uint16_t rxBdNumber

Receive buffer descriptor number.

uint16_t txBdNumber

Transmit buffer descriptor number.

uint16_t rxBuffSizeAlign

Aligned receive data buffer size.

uint16_t txBuffSizeAlign

Aligned transmit data buffer size.

volatile *enet_rx_bd_struct_t* *rxBdStartAddrAlign

Aligned receive buffer descriptor start address: should be non-cacheable.

volatile *enet_tx_bd_struct_t* *txBdStartAddrAlign

Aligned transmit buffer descriptor start address: should be non-cacheable.

uint8_t *rxBufferAlign

Receive data buffer start address.

uint8_t *txBufferAlign

Transmit data buffer start address.

bool rxMaintainEnable

Receive buffer cache maintain.

bool txMaintainEnable

Transmit buffer cache maintain.

enet_frame_info_t *txFrameInfo

Transmit frame information start address.

struct __enet_intcoalesce_config

#include <fsl_enet.h> Defines the interrupt coalescing configure structure.

Public Members

uint8_t txCoalesceFrameCount[1]

Transmit interrupt coalescing frame count threshold.

uint16_t txCoalesceTimeCount[1]

Transmit interrupt coalescing timer count threshold.

uint8_t rxCoalesceFrameCount[1]

Receive interrupt coalescing frame count threshold.

uint16_t rxCoalesceTimeCount[1]

Receive interrupt coalescing timer count threshold.

struct __enet_avb_config

#include <fsl_enet.h> Defines the ENET AVB Configure structure.

This is used for to configure the extended ring 1 and ring 2.

- a. The classification match format is $(CMP3 \ll 12) \mid (CMP2 \ll 8) \mid (CMP1 \ll 4) \mid CMP0$. composed of four 3-bit compared VLAN priority field cmp0~cmp3, cm0 ~ cmp3 are used in parallel.

If CMP1,2,3 are not unused, please set them to the same value as CMP0.

- a. The idleSlope is used to calculate the Band Width fraction, $BW \text{ fraction} = 1 / (1 + 512/idleSlope)$. For avb configuration, the BW fraction of Class 1 and Class 2 combined must not exceed 0.75.

Public Members

uint16_t rxClassifyMatch[1 - 1]

The classification match value for the ring.

enet_idle_slope_t idleSlope[1 - 1]

The idle slope for certian bandwidth fraction.

struct __enet_config

#include <fsl_enet.h> Defines the basic configuration structure for the ENET device.

Note:

- a. macSpecialConfig is used for a special control configuration, A logical OR of "enet_special_control_flag_t". For a special configuration for MAC, set this parameter to 0.
- b. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO 3 - 192 bytes written to TX FIFO The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
- c. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULLL and the maximum is 0xFF. If the end of the frame is stored in FIFO and the frame size if smaller than the

txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.

- d. When “kENET_ControlFlowControlEnable” is set in the macSpecialConfig, ensure that the pauseDuration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.
- e. When “kENET_ControlStoreAndFwdDisabled” is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
- f. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The “enet_tx_accelerator_t” and “enet_rx_accelerator_t” are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.
- g. The intCoalesceCfg can be used in the rx or tx enabled cases to decrease the CPU loading.

Public Members

uint32_t macSpecialConfig

Mac special configuration. A logical OR of “enet_special_control_flag_t”.

uint32_t interrupt

Mac interrupt source. A logical OR of “enet_interrupt_enable_t”.

uint16_t rxMaxFrameLen

Receive maximum frame length.

enet_mii_mode_t miiMode

MII mode.

enet_mii_speed_t miiSpeed

MII Speed.

enet_mii_duplex_t miiDuplex

MII duplex.

uint8_t rxAccelerConfig

Receive accelerator, A logical OR of “enet_rx_accelerator_t”.

uint8_t txAccelerConfig

Transmit accelerator, A logical OR of “enet_rx_accelerator_t”.

uint16_t pauseDuration

For flow control enabled case: Pause duration.

uint8_t rxFifoEmptyThreshold

For flow control enabled case: when RX FIFO level reaches this value, it makes MAC generate XOFF pause frame.

uint8_t rxFifoStatEmptyThreshold

For flow control enabled case: number of frames in the receive FIFO, independent of size, that can be accept. If the limit is reached, reception continues and a pause frame is triggered.

uint8_t rxFifoFullThreshold

For store and forward disable case, the data required in RX FIFO to notify the MAC receive ready status.

uint8_t txFifoWatermark

For store and forward disable case, the data required in TX FIFO before a frame transmit start.

enet_intcoalesce_config_t *intCoalesceCfg

If the interrupt coalesce is not required in the ring n(0,1,2), please set to NULL.

uint8_t ringNum

Number of used rings. default with 1 — single ring.

enet_rx_alloc_callback_t rxBuffAlloc

Callback function to alloc memory, must be provided for zero-copy Rx.

enet_rx_free_callback_t rxBuffFree

Callback function to free memory, must be provided for zero-copy Rx.

enet_callback_t callback

General callback function.

void *userData

Callback function parameter.

struct __enet_tx_bd_ring

#include <fsl_enet.h> Defines the ENET transmit buffer descriptor ring/queue structure.

Public Members

volatile enet_tx_bd_struct_t *txBdBase

Buffer descriptor base address pointer.

uint16_t txGenIdx

The current available transmit buffer descriptor pointer.

uint16_t txConsumeIdx

Transmit consume index.

volatile uint16_t txDescUsed

Transmit descriptor used number.

uint16_t txRingLen

Transmit ring length.

struct __enet_rx_bd_ring

#include <fsl_enet.h> Defines the ENET receive buffer descriptor ring/queue structure.

Public Members

volatile enet_rx_bd_struct_t *rxBdBase

Buffer descriptor base address pointer.

uint16_t rxGenIdx

The current available receive buffer descriptor pointer.

uint16_t rxRingLen

Receive ring length.

struct __enet_handle

#include <fsl_enet.h> Defines the ENET handler structure.

Public Members

enet_rx_bd_ring_t rxBdRing[1]
Receive buffer descriptor.

enet_tx_bd_ring_t txBdRing[1]
Transmit buffer descriptor.

uint16_t rxBuffSizeAlign[1]
Receive buffer size alignment.

uint16_t txBuffSizeAlign[1]
Transmit buffer size alignment.

bool rxMaintainEnable[1]
Receive buffer cache maintain.

bool txMaintainEnable[1]
Transmit buffer cache maintain.

uint8_t ringNum
Number of used rings.

enet_callback_t callback
Callback function.

void *userData
Callback function parameter.

enet_tx_dirty_ring_t txDirtyRing[1]
Ring to store tx frame information.

bool txReclaimEnable[1]
Tx reclaim enable flag.

enet_rx_alloc_callback_t rxBuffAlloc
Callback function to alloc memory for zero copy Rx.

enet_rx_free_callback_t rxBuffFree
Callback function to free memory for zero copy Rx.

uint8_t multicastCount[64]
Multicast collisions counter

uint32_t enetClock
The clock of enet peripheral, to caculate core cycles for PTP timestamp.

uint32_t tsDelayCount
The count of core cycles for PTP timestamp capture delay.

struct __enet_buffer_struct
#include <fsl_enet.h>

Public Members

void *buffer
The buffer store the whole or partial frame.

uint16_t length
The byte length of this buffer.

struct __enet_rx_frame_attribute_struct
#include <fsl_enet.h>

Public Members

bool promiscuous

This frame is received because of promiscuous mode.

```
struct __enet_rx_frame_struct  
#include <fsl_enet.h>
```

Public Members

enet_buffer_struct_t *rxBuffArray
Rx frame buffer structure.

uint16_t totLen
Rx frame total length.

enet_rx_frame_attribute_t rxAttribute
Rx frame attribute structure.

enet_rx_frame_error_t rxFrameError
Rx frame error.

```
struct __enet_tx_frame_struct  
#include <fsl_enet.h>
```

Public Members

enet_buffer_struct_t *txBuffArray
Tx frame buffer structure.

uint32_t txBuffNum
Buffer number of this Tx frame.

void *context
Driver reclaims and gives it in Tx over callback, usually store network packet header.

2.13 EPDC: Electrophoretic Display Controller

```
static inline void EPDC_EnableInterrupts(EPDC_Type *base, uint8_t interrupts)  
Enables non LUT complete EPDC interrupt source.
```

Parameters

- base – EPDC base pointer
- interrupts – The interrupt source mask, can be a single source or several sources in `_epdc_interrupt_enable` ORed together.

```
static inline void EPDC_DisableInterrupts(EPDC_Type *base, uint8_t interrupts)  
Diabls non LUT complete EPDC interrupt source.
```

Parameters

- base – EPDC base pointer
- interrupts – The interrupt source mask, can be a single source or several sources in `_epdc_interrupt_enable` ORed together.

static inline uint8_t EPDC_GetEnabledInterrupts(EPDC_Type *base)

Gets all the enabled non LUT complete interrupt sources.

Parameters

- base – EPDC base pointer

Returns

The interrupt source mask, can be a single source or several sources in _epdc_interrupt_enable ORed together.

static inline uint16_t EPDC_GetStatusFlags(EPDC_Type *base)

Gets the EPDC status flags.

Parameters

- base – EPDC base pointer

Returns

the mask of status flags, can be a single flag or several flags in _epdc_status_flags ORed together.

static inline void EPDC_ClearStatusFlags(EPDC_Type *base, uint16_t statusFlags)

Clears the EPDC status flags.

Parameters

- base – EPDC base pointer
- statusFlags – The status flag mask, can be a single flag or several flags in _epdc_status_flags ORed together.

static inline void EPDC_EnableLutCompleteInterrupts(EPDC_Type *base, uint64_t interrupts)

Enables LUT complete interrupt source.

Parameters

- base – EPDC base pointer
- interrupts – The interrupt source mask of single source or several sources up to all 64 interrupt sources. If user wants to enable the complete interrupt for LUT17, set the bit 17 in this param.

static inline void EPDC_DisableLutCompleteInterrupts(EPDC_Type *base, uint64_t interrupts)

Disables LUT complete interrupt source.

Parameters

- base – EPDC base pointer
- interrupts – The interrupt source mask of single source or several sources up to all 64 interrupt sources. If user wants to disable the complete interrupt for LUT17, set the bit 17 in this param.

static inline uint64_t EPDC_GetEnabledLutCompleteInterrupts(EPDC_Type *base)

Gets all the enabled LUT complete interrupts.

Parameters

- base – EPDC base pointer

Returns

The interrupt source mask, can be a single source or several sources up to all 64 interrupt sources. If complete interrupt for LUT17 is enabled, bit 17 is set.

static inline uint64_t EPDC_GetLutCompleteStatusFlags(EPDC_Type *base)

Gets the LUT complete status flags.

Parameters

- base – EPDC base pointer

Returns

the mask of status flags, can be a single flag or several flags up to all 64 interrupt flags. If LUT17 has completed, bit 17 is set.

`static inline void EPDC_ClearLutCompleteStatusFlags(EPDC_Type *base, uint64_t statusFlags)`

Clears the LUT complete status flags.

Parameters

- base – EPDC base pointer
- the – mask of status flags, can be a single flag or several flags up to all 64 interrupt flags. If user wants to clear LUT17 completion, set bit 17 in this param.

`void EPDC_ResetToInit(EPDC_Type *base)`

Resets the EPDC to initialized state.

Parameters

- base – EPDC base pointer

`void EPDC_Init(EPDC_Type *base)`

Initializes the EPDC.

This function enables the EPDC peripheral clock, and resets the EPDC registers to default status.

Parameters

- base – EPDC base pointer

`void EPDC_Deinit(EPDC_Type *base)`

De-initializes the EPDC.

This function disables the EPDC peripheral clock.

Parameters

- base – EPDC base pointer

`status_t EPDC_InitDisplay(EPDC_Type *base, const epdc_display_config_t *config)`

Initializes the EPDC display parameters.

Parameters

- base – EPDC base pointer
- config – Pointer to EPDC display configuration structure

Return values

- kStatus_Success – Successfully initialized the display parameters
- kStatus_InvalidArgument – parameter(s) is(are) not valid

`status_t EPDC_ConfigFifo(EPDC_Type *base, const epdc_fifo_config_t *config)`

Configures the FIFO control parameters, including panic function and pre-fill level.

Parameters

- base – EPDC base pointer
- config – Pointer to EPDC FIFO control configuration structure

Return values

- kStatus_Success – Successfully configured the FIFO
- kStatus_InvalidArgument – parameter(s) is(are) not valid

static inline void EPDC_EnableFifoPanic(EPDC_Type *base, bool enable)

Enables/disables the panic function for FIFO control.

Parameters

- base – EPDC base pointer
- enable – true to enable panic function, false to disable

status_t EPDC_ConfigTCE(EPDC_Type *base, const *epdc_tce_config_t* *config)

Configures the TCE parameters before initiating the panel update.

Note: Make sure to configure the display parameters by EPDC_InitDisplay before calling this API.

Parameters

- base – EPDC base pointer
- config – Pointer to EPDC TCE control configuration structure

Return values

- kStatus_Success – Successfully configured the TCE
- kStatus_InvalidArgument – parameter(s) is(are) not valid

void EPDC_ConfigPigeon(EPDC_Type *base, uint8_t signalNum, const *epdc_pigeon_config_t* *config)

Configures the signal output format for signals that support pigeon mode.

Note: Refer to the soc reference manual to check the configurable pigeon signal. No need to call this API if not using the pigeon mode.

Parameters

- base – EPDC base pointer
- signalNum – The number of the pigeon signal to be configured
- config – Pointer to EPDC pigeon mode configuration structure

void EPDC_ConfigPigeonCycle(EPDC_Type *base, const *epdc_pigeon_cycle_config_t* *config)

Configures the cycle's period.

When selecting any of the pclk/line/frame cycle as global counter, call this API first to set the proper cycle period. The configuration is shared for all pigeon signals.

Parameters

- base – EPDC base pointer
- config – Pointer to EPDC pigeon mode cycle configuration structure

status_t EPDC_GetNextAvailableLUT(EPDC_Type *base, uint8_t *lutNum)

Gets the number of the next available LUT.

Parameters

- base – EPDC base pointer
- lutNum – Pointer to LUT number

Return values

- kStatus_Success – Successfully got the LUT number

- kStatus_Fail – no valid LUT present, all LUTs are busy

static inline void EPDC_ConfigureWaveform(EPDC_Type *base, uint8_t waveformMode)

Configures the waveform mode and initiate the panel update.

Parameters

- base – EPDC base pointer
- waveformMode – Waveform mode

status_t EPDC_UpdateDisplay(EPDC_Type *base, epdc_update_config_t *config)

Initiates a panel display update.

Parameters

- base – EPDC base pointer
- config – Pointer to update configuration structure

Return values

- kStatus_Success – Successfully initiate an update
- kStatus_Fail – update failed due to busy working buffer or LUT
- kStatus_InvalidArgument – parameter(s) is(are) not valid

void EPDC_SetAutowaveMap(EPDC_Type *base, const epdc_automap_t *config)

Sets one map between a grey level and its waveform mode.

Call this API multiple times until all the maps are set.

Parameters

- base – EPDC base pointer
- config – Pointer to autowave configuration structure

void EPDC_SetGpioOutput(EPDC_Type *base, const epdc_gpio_config_t *config)

Sets the GPIO output value for BDR(0,1), PWR_CTRL(0,1,2,3), PWR_COM and PWR_WAKE pins.

Parameters

- base – EPDC base pointer
- config – Pointer to GPIO configuration structure

void EPDC_GetCollisionStatus(EPDC_Type *base, epdc_collision_status_t *status)

Gets the status of the collision.

Parameters

- base – EPDC base pointer
- status – Pointer to collision status structure

FSL_EPDC_DRIVER_VERSION

enum __epdc_interrupt_enable

EPDC non LUT complete interrupts.

Note: These enumerations are meant to be OR'd together to form a bit mask. EPDC also has 64 independent LUT complete interrupts.

Values:

enumerator kEPDC_WbCompleteInterruptEnable

Working buffer complete interrupt.

enumerator kEPDC_LutCollisionDetetionInterruptEnable

Collision detection interrupt for all LUTs.

enumerator kEPDC_FifoUnderRunInterruptEnable

Pixel FIFO underrun interrupt.

enumerator kEPDC_FrameCompleteInterruptEnable

Current frame is complete and in vertical blanking period interrupt.

enumerator kEPDC_AxiBusErrorInterruptEnable

AXI bus error occur interrupt.

enumerator kEPDC_TceIdleInterruptEnable

TCE finished frame scan and in isle state interrupt.

enumerator kEPDC_UpdateCompleteInterruptEnable

Update complete interrupt.TODO

enumerator kEPDC_PowerInterruptEnable

Power interrupt.TODO

enum _epdc_status_flags

EPDC non LUT complete status flags.

Note: These enumerations are meant to be OR'd together to form a bit mask. EPDC also has 64 independent LUT complete interrupt status.

Values:

enumerator kEPDC_WbBusyFlag

Working buffer process is busy and cannot accept new update requests.

enumerator kEPDC_LutBusyFlag

All the LUTs ary busy.

enumerator kEPDC_LutUnderrunFlag

Not enough time for active luts read during blanking period.

enumerator kEPDC_UpdateBufferVoidFlag

Update buffer is void indicating no pixels need updating or all pixels were collided.

enumerator kEPDC_WbCompleteInterruptFlag

Working buffer complete interrupt flag.

enumerator kEPDC_LutCollisionDetetionInterruptFlag

Collision detection interrupt flag for all LUTs.

enumerator kEPDC_FifoUnderRunInterruptFlag

Pixel FIFO underrun interrupt flag.

enumerator kEPDC_FrameCompleteInterruptFlag

Interrupt flag indicating current frame is complete and in vertical blanking period.

enumerator kEPDC_AxiBusErrorInterruptFlag

AXI bus error occur interrupt flag.

enumerator kEPDC_TceIdleInterruptFlag

Interrupt flag indicationg TCE has finished frame scan and in isle state.

enumerator kEPDC_UpdateCompleteInterruptFlag

Update complete interrupt flag.TODO

enumerator kEPDC_PowerInterruptFlag

Power interrupt flag.TODO

enum _epdc_pixel_format

EPDC TFT pixel format.

Values:

enumerator kEPDC_Pixel2bit

enumerator kEPDC_Pixel2bitVcom

enumerator kEPDC_Pixel4bit

enumerator kEPDC_Pixel4bitVcom

enum _epdc_wb_type

EPDC working buffer type.

Values:

enumerator kEPDC_WbInternal

Internal Working Buffer written by EPDC itself.

enumerator kEPDC_WbWaveform

Working buffer is actually holding waveform.

enumerator kEPDC_Wb16bit

16bit working buffer written by PXP or CPU.

enumerator kEPDC_Wb32bit

32bit working buffer written by PXP or CPU.

enum _epdc_data_swizzle

EPDC data swizzle type for update date or LUT data.

Values:

enumerator kEPDC_NoSwap

Little endian, no byte swapping.

enumerator kEPDC_SwapAllBytes

Big endian, byte 0|1|2|3 -> 3|2|1|0.

enumerator kEPDC_SwapHalfWord

byte 0|1|2|3 -> 2|3|0|1.

enumerator kEPDC_SwapWithinHalfWord

byte 0|1|2|3 -> 1|0|3|2.

enum _epdc_tce_vcom_mode

The VCOM value configuration.

Values:

enumerator kEPDC_VcomManual

VCOM value is set manually, call EPDC_SetGpioOutput to set the VCOM value output.

enumerator kEPDC_VcomAuto

VCOM value is used from waveform.

enum _epdc_data_output_mode

The data output mode using DATA0 - DATA15.

Values:

enumerator kEPDC_Sdr8bit

Data driven only on positive edge of SDCLK. Connect to 8-bit SDDO.

enumerator kEPDC_Sdr16bit

Data driven only on positive edge of SDCLK. Connect to 8-bit SDDO.

enumerator kEPDC_Ddr8bit

Data driven on both edge of SDCLK. Connect to 8-bit SDDO.

enumerator kEPDC_Ddr16bit

Data driven on both edge of SDCLK. Connect to 16-bit SDDO.

enumerator kEPDC_Lvds

DATA[15:8] is for differential signalling. Requires data driven on both edge of SDCLK and connect to 16-bit SDDO.

enumerator kEPDC_SdrDualScan

Data driven only on positive edge of SDCLK. DATA[7:0] and DATA[15:8] drive the upper/lower half of the panel.

enumerator kEPDC_DdrDualScan

Data driven on both edge of SDCLK. DATA[7:0] and DATA[15:8] drive the upper/lower half of the panel.

enum _epdc_dualscan_direction

The vertical scan direction in dual scan mode.

Values:

enumerator kEPDC_DualScanTopToBottom

Scan this region from top to bottom.

enumerator kEPDC_DualScanBottomToTop

Scan this region from bottom to top.

enum _epdc_polarity

The signal output polarity for SDCE/SDLE/SDOE in source driver and GDOE/GDSP in gate driver.

Values:

enumerator kEPDC_ActiveLow

enumerator kEPDC_ActiveHigh

enum _epdc_data_reverse

The data reverse configuration for source driver configuration.

Values:

enumerator kEPDC_DataUnchanged

enumerator kEPDC_DataReversed

Reverse the pixel order on SDDO, eg: 2B/pixel on 8-bit bus, P3,P2,P1,P0->P0,P1,P2,P3.

enum _epdc_shift_direction

Shift direction for SDSHR/GDRL signal for source/gate driver configuration.TODO.

Values:

enumerator kEPDC_ShiftRight

enumerator kEPDC_ShiftLeft

enum _epdc_signal_delay

SDCLK/SDLE signal delay.

Values:

enumerator kEPDC_NoDelay

No delay.

enumerator kEPDC_Delay1Cycle

Delay 1 SDCLK cycle.

enumerator kEPDC_Delay2Cycle

Delay 2 SDCLK cycle.

enumerator kEPDC_Delay3Cycle

Delay 3 SDCLK cycle.

enum _epdc_gdoe_mode

The output mode for GDOE signal.

Values:

enumerator kEPDC_GDOEActiveBesideSync

GDOE is driven at all times during the frame-scan except FRAME_SYNC.

enumerator kEPDC_GDOEActiveDelayedSck

GDOE is driven as a delayed version of GDCLK, and only active during the frame-data time.

enum _epdc_gdsp_mode

The output mode for GDSP signal.

Values:

enumerator kEPDC_GDSPActiveOnBegin

GDSP is driven on the first line clock of FRAME_BEGIN for one GDCLK period, and can be shifted using the GDSP_OFFSET control.

enumerator kEPDC_GDSPActiveDuringSync

GDSP is active during the entire FRAME_SYNC, and GDSP_OFFSET has no effect.

enum _epdc_pigeon_localCounter_tickEvent

Local counter tick event.

Values:

enumerator kEPDC_TickOnPclk

Counter tick on pixel clock.

enumerator kEPDC_TickOnLineStart

Counter tick on line start pulse.

enumerator kEPDC_TickOnFrameStart

Counter tick on frame start pulse.

enumerator kEPDC_TickOnAnotherSignal

Use another signal as tick event.

enum _epdc_pigeon_globalCounter

Global counter select for pigeon signal's local counter start event. If selecting any of the pclk/line/frame cycle, need to call EPDC_ConfigPigeonCycle to set the cycle period first. The cycle configuration applies to all pigeon signals.

Values:

enumerator kEPDC_PclkCounterReset

Use pclk counter as global counter, counter reset when state changes.

enumerator kEPDC_PclkCycleReset

Use pclk cycle as global counter, counter reset when state changes. Refer to epdc_pigeon_cycle_config_t for the definition of cycle.

enumerator kEPDC_LineCounterReset

Use line counter as global counter, counter reset when state changes.

enumerator kEPDC_LineCycleReset

Use line cycle as global counter, counter reset when state changes. Refer to epdc_pigeon_cycle_config_t for the definition of cycle.

enumerator kEPDC_FrameCounter

Use frame counter as global counter.

enumerator kEPDC_FrameCycle

Use frame cycle as global counter. Refer to epdc_pigeon_cycle_config_t for the definition of cycle.

enumerator kEPDC_PclkCounter

Use pclk counter as global counter.

enumerator kEPDC_LineCounter

Use line counter as global counter.

enum _epdc_pigeon_localCounter_startEvent

The event for the pigeon signal's local counter to start ticking. Chose one or more frame events and one or more line events, OR them together to form the final event. eg: Use (kEPDC_FrameData | kEPDC_LineBegin), then the local counter will start ticking when line begin starts on the frame data phase.

Values:

enumerator kEPDC_FrameSync

enumerator kEPDC_FrameBegin

enumerator kEPDC_FrameData

enumerator kEPDC_FrameEnd

enumerator kEPDC_LineSync

enumerator kEPDC_LineBegin

enumerator kEPDC_LineData

enumerator kEPDC_LineEnd

enum _epdc_pigeon_output_logic

Signal output logic .

Values:

enumerator kEPDC_NoLogic

No logic operation, use the chosen another signal directly as the output.

enumerator kEPDC_SignalAndAnother

Use the pigeon signal AND this chosen another signal to form the final output.

enumerator kEPDC_SignalOrAnother

Use the pigeon signal OR this chosen another signal to form the final output.

enumerator kEPDC_StartPointAndAnother

Use local counter's start point AND this chosen another signal to form the final output.TODO

enum _epdc_waveform_mode

Waveform mode.

Values:

enumerator kEPDC_WaveformInit

Panel initialization. Completely erase the display and leave it in the white state.

enumerator kEPDC_WaveformDU

Direct update. Very fast, non-flashy, transitions from any graytone to black or white only.

enumerator kEPDC_WaveformGC16

Grayscale clearing. High image quality, has 16 unique gray levels.

enumerator kEPDC_WaveformGL16

Used to update scattered content on a white background with reduced flash, has 16 unique gray levels.

enumerator kEPDC_WaveformGLR16

Similar to GL16, but if a separately-supplied image preprocessing algorithm is used, display quality is improved when transitions to pixel states 29 and 31.

enumerator kEPDC_WaveformGLD16

Similar to GL16, but if a separately-supplied image preprocessing algorithm is used, display quality is improved when transitions to pixel states 29 and 31, with lighter flash of the background compared to GC16 mode, and reduce image artifacts even more compared to the GLR16 mode.

enumerator kEPDC_WaveformA2

Very fast, non-flashy, transitions from black or white to black or white only.

enumerator kEPDC_WaveformDU4

Direct update 4. Very fast, non-flashy, from any graytone to gray tones 1,6,11,16 represented by pixel states [0 10 20 30].

typedef enum _epdc_pixel_format epdc_pixel_format_t

EPDC TFT pixel format.

typedef enum _epdc_wb_type epdc_wb_type_t

EPDC working buffer type.

typedef enum _epdc_data_swizzle epdc_data_swizzle_t

EPDC data swizzle type for update date or LUT data.

typedef struct _epdc_display_config epdc_display_config_t

EPDC display configuration.

typedef struct _epdc_fifo_config epdc_fifo_config_t

EPDC FIFO control configuration.

typedef enum *_epdc_tce_vcom_mode* epdc_tce_vcom_mode_t

The VCOM value configuration.

typedef enum *_epdc_data_output_mode* epdc_data_output_mode_t

The data output mode using DATA0 - DATA15.

typedef enum *_epdc_dualscan_direction* epdc_dualscan_direction_t

The vertical scan direction in dual scan mode.

typedef enum *_epdc_polarity* epdc_polarity_t

The signal output polarity for SDCE/SDLE/SDOE in source driver and GDOE/GDSP in gate driver.

typedef enum *_epdc_data_reverse* epdc_data_reverse_t

The data reverse configuration for source driver configuration.

typedef enum *_epdc_shift_direction* epdc_shift_direction_t

Shift direction for SDSHR/GDRL signal for source/gate driver configuration.TODO.

typedef enum *_epdc_signal_delay* epdc_signal_delay_t

SDCLK/SDLE signal delay.

typedef struct *_epdc_sd_config* epdc_sd_config_t

EPDC source driver configuration. Source driver controls the timing for horizontal scan.

typedef enum *_epdc_gdoe_mode* epdc_gdoe_mode_t

The output mode for GDOE signal.

typedef enum *_epdc_gdsp_mode* epdc_gdsp_mode_t

The output mode for GDSP signal.

typedef struct *_epdc_gd_config* epdc_gd_config_t

EPDC gate driver configuration. Gate driver controls the timing for vertical scan.

typedef struct *_epdc_scan_config* epdc_scan_config_t

EPDC scan configuration. Including horizontal scan(line timing) and vertical scan(frame timing).

typedef struct *_epdc_tce_config* epdc_tce_config_t

EPDC TCE(timing controller engine) configuration.

typedef enum *_epdc_pigeon_localCounter_tickEvent* epdc_pigeon_localCounter_tickEvent_t

Local counter tick event.

typedef enum *_epdc_pigeon_globalCounter* epdc_pigeon_globalCounter_t

Global counter select for pigeon signal's local counter start event. If selecting any of the pclk/line/frame cycle, need to call EPDC_ConfigPigeonCycle to set the cycle period first. The cycle configuration applies to all pigeon signals.

typedef enum *_epdc_pigeon_output_logic* epdc_pigeon_output_logic_t

Signal output logic .

typedef struct *_epdc_pigeon_config* epdc_pigeon_config_t

Signal pigeon mode configuration. Each pigeon signal has one local counter with a configurable start/tick event. The counter value is compared to setCount/clearCount for signal assertion/de-assertion.

typedef struct *_epdc_pigeon_cycle_config* epdc_pigeon_cycle_config_t

The picle/line/frame cycle definition for global counter, shared by all pigeon signals.

typedef enum *_epdc_waveform_mode* epdc_waveform_mode_t

Waveform mode.

```
typedef struct _epdc_update_config epdc_update_config_t
    Panel update configuration.

typedef struct _epdc_autowave_map epdc_autowave_map_t
    Autowave map configuration.

typedef struct _epdc_gpio_config epdc_gpio_config_t
    GPIO pin output configuration.

typedef struct _epdc_collision_status epdc_collision_status_t
    Update collision status structure.

struct _epdc_display_config
    #include <fsl_epdc.h> EPDC display configuration.
```

Public Members

```
uint32_t waveformAddr
    64-bit aligned waveform table address.

uint32_t wbAddr
    64-bit aligned working buffer address.

uint32_t tempIdx
    Temperature index, used by the EPDC to access the correct temperature compensated waveform.

uint16_t resX
    Horizontal resolution in pixels.

uint16_t resY
    Vertical resolution in pixels.

uint8_t defaultTftPixelValue
    The source-driver voltage value for either partial updates where a pixel has not changed or for any part of the screen which is not being updated during active frame scans.

epdc_pixel_format_t pixelFormat
    How many bits of the SDDO bus are required per pixel. This field must be consistent with the waveform and panel architecture.

epdc_wb_type_t wbType
    Working buffer type.

epdc_data_swizzle_t updSwizzle
    How to swap the bytes for the UPD data before the WB construction.

epdc_data_swizzle_t lutSwizzle
    How to swap the bytes for the LUT data before store to LUTRAM.

struct _epdc_fifo_config
    #include <fsl_epdc.h> EPDC FIFO control configuration.
```

Public Members

```
uint32_t lowLevel
    Low level for panic function.
```

uint32_t highLevel

High level for panic function.

uint32_t prefillLevel

EPDC pre-fill pixel FIFO to this level before TCE starts refresh operations. Must be less than VERTICAL x HORIZONTAL / 4.

uint32_t __pad0__

Reserved.

uint32_t panicEnable

Enable epdc_panic output signal for priority elevation. Signal asserts when data in FIFO is below the low level and de-asserts when above the high level. When asserted, EPDC transaction requests have elevated priority at SoC-level bus arbitration.

struct __epdc_sd_config

#include <fsl_epdc.h> EPDC source driver configuration. Source driver controls the timing for horizontal scan.

Public Members

uint32_t sdcePol

SDCE polarity. Refer epdc_polarity_t.

uint32_t sdlePol

SDLE polarity. Refer epdc_polarity_t.

uint32_t sdoePol

SDOE polarity, including SDOE/SDOEZ/SDOED. Refer epdc_polarity_t.

uint32_t sdclk_delay

Additional delay to the SDCLK generation. Refer epdc_signal_delay_t.

uint32_t __pad0__

Reserved.

uint32_t sdclk_invert

Invert the SDCLK signal.

uint32_t sdle_delay

Additional delay to the SDLE generation. Refer epdc_signal_delay_t.

uint32_t __pad1__

Reserved.

uint32_t dataInvert

Invert the polarity of each SDDO bit, eg: 0xAAAA -> 0x5555.

uint32_t pixelReverse

Reverse the pixel order on SDDO, eg: 2B/pixel on 8-bit bus, P3,P2,P1,P0->P0,P1,P2,P3. Refer epdc_data_reverse_t.

uint32_t sdceCount

The total count of source driver IC chip-enable pins. Make sure the horizontal resolution can be evenly divided by it.

uint32_t shiftDir

The value of the SDSHR output signal controlling shift direction. Refer epdc_shift_direction_t.

uint32_t clockHoldEnable

Enable to hold the SDCLK low firing line begin phase.

uint32_t __pad2__

Reserved.

uint32_t sdoezDelay

Number of PIXCLK cycles from SDLE falling edge to SDOEZ rising.

uint32_t sdoezWidth

Number of PIXCLK cycles from SDOEZ high to SDOEZ falling.

uint32_t sdoedDelay

Number of PIXCLK cycles from SDOEZ low to SDOED rising.

uint32_t sdoedWidth

Number of PIXCLK cycles from SDOED high to SDOED falling.

struct __epdc_gd_config

#include <fsl_epdc.h> EPDC gate driver configuration. Gate driver controls the timing for vertical scan.

Public Members

epdc_shift_direction_t shiftDir

The value of the GDRL output signal controlling gate driver pulse shift direction.

epdc_gdoe_mode_t gdoeMode

GDOE(gate driver output enable) signal mode.

epdc_gdsp_mode_t gdspMode

GDSP(gate driver start pulse) signal mode.

epdc_polarity_t gdoePol

GDOE polarity.

epdc_polarity_t gdspPol

GDSP polarity.

uint16_t gdoeOffset

Delay GDOE in the terms of PIXCLK.

uint16_t gdspOffset

Delay GDSP in the terms of PIXCLK, only works when GDSP mode is kEPDC_GDSPActiveOnBegin.

uint16_t gdClkOffset

Delay GDCLK from the line time in the terms of PIXCLK.

uint16_t gdClkHigh

GDCLK high-pulse width in PIXCLK. The GDCLK cycle(high+low) always equal to line timing.

struct __epdc_scan_config

#include <fsl_epdc.h> EPDC scan configuration. Including horizontal scan(line timing) and vertical scan(fram timing).

Public Members

uint16_t lineSync

Number of PIXCLK cycles for line sync duration.

uint16_t lineBegin

Number of PIXCLK cycles for line begin duration.

uint16_t lineEnd

Number of PIXCLK cycles for line end duration.

uint8_t frameSync

Number of lines for frame sync duration.

uint8_t frameBegin

Number of lines for frame begin duration.

uint8_t frameEnd

Number of lines for frame end duration.

struct __epdc_tce_config

#include <fsl_epdc.h> EPDC TCE(timing controller engine) configuration.

Public Members

uint32_t tceWbAddr

Working buffer address for TCE.

uint16_t vscanHoldoff

9-bit max. Hold off the vertical scan timing during the blanking period to allow more smaller updates to be processed and included in the frame scan. Note the remainder of the blanking period is reserved for pre-filling the TCE pixel FIFOs so make sure enough time is left to prevent FIFO underrun.

uint8_t vcomVal

2-bit max. The VCOM value when mode is manual.

epdc_tce_vcom_mode_t vcomMode

The method to driver VCOM signal.

epdc_data_output_mode_t outMode

The data output mode.

epdc_dualscan_direction_t upperScanDirection

The scan direction of the upper half in dual scan mode.

epdc_dualscan_direction_t lowerScanDirection

The scan direction of the lower half in dual scan mode.

epdc_sd_config_t sdConfig

Source driver configuration

epdc_gd_config_t gdConfig

Gate driver configuration

epdc_scan_config_t scanConfig

Scan configuration

struct __epdc_pigeon_config

#include <fsl_epdc.h> Signal pigeon mode configuration. Each pigeon signal has one local counter with a configurable start/tick event. The counter value is compared to setCount/clearCount for signal assertion/de-assertion.

Public Members

uint32_t enablePigeon

Enable pigeon mode for this signal.

uint32_t polarity

Polarty of the signal output.

uint32_t localCounterTickEvent

The tick event selection for local counter. Refer epdc_pigeon_localCounter_tickEvent_t.

uint32_t offset

The offset of pclk unit, to let the signal come out earlier/later.TODO

uint32_t globalCounterSelect

To use a more accurate local counter start event than localCounterStartEvent, a global counter combined with startCount can be used. Refer epdc_pigeon_globalCounter_t.

uint32_t startCount

When the value of the selected global counter matches the count, the local counter starts.

uint32_t localCounterStartEvent

The event to start the local counter. Refer _epdc_pigeon_localCounter_startEvent.

uint32_t setCount

Assert signal when local counter matches this value.

uint32_t clearCount

De-assert signal when local counter matches this value.

uint32_t outputLogic

When localCounterTickEvent is kEPDC_TickOnOtherSignal, use this logic to form the final output. Refer epdc_pigeon_output_logic_t.

uint32_t anotherSignalNum

When localCounterTickEvent is kEPDC_TickOnOtherSignal, use the pigeon signal of this number to form the tick event.

uint32_t __pad0__

Reserved.

struct _epdc_pigeon_cycle_config

#include <fsl_epdc.h> The pixle/line/frame cycle definition for global counter, shared by all pegin signals.

Public Members

uint32_t lineCount

If chosing kEPDC_LineCycleReset, n lines in a frame are considered as one cycle, when the period count matches startCount global counter starts ticking.

uint32_t __pad0__

Reserved.

uint32_t pixCount

If chosing kEPDC_PclkCycleReset, n pixles in a line are considered as one cycle, when the period count matches startCount global counter starts ticking.

uint32_t __pad1__

Reserved.

uint32_t frameCount

If choosing kEPDC_FrameCycle, n frames are considered as one cycle, when the period count matches startCount global counter starts ticking.

uint32_t __pad2__

Reserved.

uint32_t maxframeCycle

The max cycles of frame cycle counter, frame cycle counter resets when reaches this value.

uint32_t __pad3__

Reserved.

struct __epdc_update_config

#include <fsl_epdc.h> Panel update configuration.

Public Members

uint32_t bufferAddress

Address pointing to the update region which will be processed into the working buffer. If stride is zero(disabled), the address must start and end on the 8-byte aligned addresses.

uint32_t stride

Line stride.

uint32_t coordinateX

X coordinate of the new update region.

uint32_t __pad0__

Reserved.

uint32_t coordinateY

Y coordinate of the new update region.

uint32_t __pad1__

Reserved.

uint32_t width

The width of the new update region in pixels.

uint32_t __pad2__

Reserved.

uint32_t height

The height of the new update region in pixels.

uint32_t __pad3__

Reserved.

uint32_t cpValue

The CP value if using fixed CP.

uint32_t npValue

The NP value if using fixed NP.

uint32_t __pad4__

Reserved.

uint32_t fixedCpEnable

Enable fixed CP, need to set fixedEnable first.

uint32_t fixedNpEnable

Enable fixed NP, need to set fixedEnable first.

uint32_t fullUpdateEnable

Enable full update.

uint32_t dryRunEnable

Enable dryrun which runs an update to obtain information without performing any real action.

uint32_t autoWaveEnable

Enable automatical waveform mode selection. After completes the update buffer processing, EPDC reports the minimal grey level, retrieves the mapped waveform mode and program it into UPD_CTRL[WAVEFORM_MODE]. To use autowave the waveform map should be programmed in advance.

uint32_t pauseEnable

Enable pause after auto waveform selection. User can choose to check and alter the waveform mode then initiate the update by calling EPDC_ConfigureWaveform.

uint32_t voidUpdateCandleDisable

Disable the feature of cancelling void update(due to collision).

uint32_t __pad5__

Reserved.

uint32_t waveformMode

Waveform used for the update. Refer epdc_waveform_mode_t.

uint32_t lutNum

The LUT number used for the update, can use EPDC_GetNextAvailableLUT.

uint32_t __pad6__

Reserved.

uint32_t fixedEnable

Use fixed pixel values.

struct _epdc_ autowave_map

#include <fsl_epdc.h> Autowave map configuration.

Public Members

uint32_t greyLevel

Grey level.

uint32_t __pad0__

Reserved.

uint32_t waveformMode

Waveform mode. Refer @epdc_waveform_mode_t.

uint32_t __pad1__

Reserved.

struct _epdc_gpio_config

#include <fsl_epdc.h> GPIO pin output configuration.

Public Members

uint32_t bdrOutput

The BDR(0,1) pin output value.

uint32_t pwrCtrlOutput

The PWR_CTRL(0,1,2,3) pin output value.

uint32_t pwrComOutput

The PWR_COM pin output value.

uint32_t pwrWakeOutput

The PWR_WAKE pin output value.

uint32_t __pad0__

Reserved.

struct __epdc_collision_status

#include <fsl_epdc.h> Update collision status structure.

Public Members

uint16_t minX

The x offset of the ULC of the collision region.

uint16_t minY

The y offset of the ULC of the collision region.

uint16_t width

The width of the collision region.

uint16_t height

The height of the collision region.

uint64_t lutlist

64-bit LUT list of the victim LUT(s), if LUT17 is being collided, then the bit17 of the list is set.

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 FGPIO Driver

2.16 FlexCAN: Flex Controller Area Network Driver

2.17 FlexCAN Driver

```
bool FLEXCAN_IsInstanceHasFDMode(CAN_Type *base)
```

Determine whether the FlexCAN instance support CAN FD mode at run time.

Note: Use this API only if different soc parts share the SOC part name macro define. Otherwise, a different SOC part name can be used to determine at compile time whether the FlexCAN instance supports CAN FD mode or not. If need use this API to determine if CAN FD mode is supported, the FLEXCAN_Init function needs to be executed first, and then call this API and use the return to value determines whether to supports CAN FD mode, if return true, continue calling FLEXCAN_FDInit to enable CAN FD mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

return TRUE if instance support CAN FD mode, FALSE if instance only support classic CAN (2.0) mode.

status_t FLEXCAN_EnterFreezeMode(CAN_Type *base)

Enter FlexCAN Freeze Mode.

This function makes the FlexCAN work under Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

status_t FLEXCAN_ExitFreezeMode(CAN_Type *base)

Exit FlexCAN Freeze Mode.

This function makes the FlexCAN leave Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

uint32_t FLEXCAN_GetInstance(CAN_Type *base)

Get the FlexCAN instance from peripheral base address.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN instance.

bool FLEXCAN_CalculateImprovedTimingValues(CAN_Type *base, *uint32_t* bitRate, *uint32_t* sourceClock_Hz, *flexcan_timing_config_t* *pTimingConfig)

Calculates the improved timing values by specific bit Rates for classical CAN.

This function use to calculates the Classical CAN timing values according to the given bit rate. The Calculated timing values will be set in CTRL1/CBT/ENCBT register. The calculation is based on the recommendation of the CiA 301 v4.2.0 and previous version document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The classical CAN speed in bps defined by user, should be less than or equal to 1Mbps.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration.

void FLEXCAN_Init(CAN_Type *base, const *flexcan_config_t* *pConfig, uint32_t sourceClock_Hz)
 Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the *flexcan_config_t* parameters and how to call the FLEXCAN_Init function by passing in these parameters.

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc      = kFLEXCAN_ClkSrc0;
flexcanConfig.bitRate     = 1000000U;
flexcanConfig.maxMbNum    = 16;
flexcanConfig.enableLoopBack = false;
flexcanConfig.enableSelfWakeup = false;
flexcanConfig.enableIndividMask = false;
flexcanConfig.enableDoze    = false;
flexcanConfig.disableSelfReception = false;
flexcanConfig.enableListenOnlyMode = false;
flexcanConfig.timingConfig = timingConfig;
FLEXCAN_Init(CAN0, &flexcanConfig, 4000000UL);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.

bool FLEXCAN_FDCalculateImprovedTimingValues(CAN_Type *base, uint32_t bitRate, uint32_t bitRateFD, uint32_t sourceClock_Hz, *flexcan_timing_config_t* *pTimingConfig)

Calculates the improved timing values by specific bit rates for CANFD.

This function use to calculates the CANFD timing values according to the given nominal phase bit rate and data phase bit rate. The Calculated timing values will be set in CBT/ENCBT and FDCBT/EDCBT registers. The calculation is based on the recommendation of the CiA 1301 v1.0.0 document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The CANFD bus control speed in bps defined by user.
- bitRateFD – The CAN FD data phase speed in bps defined by user. Equal to bitRate means disable bit rate switching.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

void FLEXCAN_FDInit(CAN_Type *base, const *flexcan_config_t* *pConfig, uint32_t sourceClock_Hz, *flexcan_mb_size_t* dataSize, bool brs)

Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the *flexcan_config_t* parameters and how to call the FLEXCAN_FDInit function by passing in these parameters.

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc      = kFLEXCAN_ClkSrc0;
flexcanConfig.bitRate     = 1000000U;
```

(continues on next page)

(continued from previous page)

```

flexcanConfig.bitRateFD      = 2000000U;
flexcanConfig.maxMbNum      = 16;
flexcanConfig.enableLoopBack = false;
flexcanConfig.enableSelfWakeup = false;
flexcanConfig.enableIndividMask = false;
flexcanConfig.disableSelfReception = false;
flexcanConfig.enableListenOnlyMode = false;
flexcanConfig.enableDoze      = false;
flexcanConfig.timingConfig    = timingConfig;
FLEXCAN_FDIInit(CAN0, &flexcanConfig, 8000000UL, kFLEXCAN_16BperMB, true);

```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.
- dataSize – FlexCAN Message Buffer payload size. The actual transmitted or received CAN FD frame data size needs to be less than or equal to this value.
- brs – True if bit rate switch is enabled in FD mode.

void FLEXCAN_Deinit(CAN_Type *base)

De-initializes a FlexCAN instance.

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

- base – FlexCAN peripheral base address.

void FLEXCAN_GetDefaultConfig(*flexcan_config_t* *pConfig)

Gets the default configuration structure.

This function initializes the FlexCAN configuration structure to default values. The default values are as follows. flexcanConfig->clkSrc = kFLEXCAN_ClkSrc0; flexcanConfig->bitRate = 1000000U; flexcanConfig->bitRateFD = 2000000U; flexcanConfig->maxMbNum = 16; flexcanConfig->enableLoopBack = false; flexcanConfig->enableSelfWakeup = false; flexcanConfig->enableIndividMask = false; flexcanConfig->disableSelfReception = false; flexcanConfig->enableListenOnlyMode = false; flexcanConfig->enableDoze = false; flexcanConfig->enablePretendededeNetworking = false; flexcanConfig->enableMemoryErrorControl = true; flexcanConfig->enableNonCorrectableErrorEnterFreeze = true; flexcanConfig->enableTransceiverDelayMeasure = true; flexcanConfig->enableRemoteRequestFrameStored = true; flexcanConfig->payloadEndianness = kFLEXCAN_bigEndian; flexcanConfig.timingConfig = timingConfig;

Parameters

- pConfig – Pointer to the FlexCAN configuration structure.

void FLEXCAN_SetTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN classical CAN protocol timing characteristic.

This function gives user settings to classical CAN or CAN FD nominal phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetBitRate() instead.

Note: Calling FLEXCAN_SetTimingConfig() overrides the bit rate set in FLEXCAN_Init() or FLEXCAN_SetBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t bitRate_Bps)

Set bit rate of FlexCAN classical CAN frame or CAN FD frame nominal phase.

This function set the bit rate of classical CAN frame or CAN FD frame nominal phase base on FLEXCAN_CalculateImprovedTimingValues() API calculated timing values.

Note: Calling FLEXCAN_SetBitRate() overrides the bit rate set in FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRate_Bps – Bit rate in Bps.

Returns

kStatus_Success - Set CAN baud rate (only Nominal phase) successfully.

void FLEXCAN_SetFDTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN CANFD data phase timing characteristic.

This function gives user settings to CANFD data phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetFDBitRate() to set both Nominal/Data bit Rate instead.

Note: Calling FLEXCAN_SetFDTimingConfig() overrides the data phase bit rate set in FLEXCAN_FDInit()/FLEXCAN_SetFDBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetFDBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t bitRateN_Bps, uint32_t bitRateD_Bps)

Set bit rate of FlexCAN FD frame.

This function set the baud rate of FLEXCAN FD base on FLEXCAN_FDCalculateImprovedTimingValues() API calculated timing values.

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRateN_Bps – Nominal bit Rate in Bps.
- bitRateD_Bps – Data bit Rate in Bps.

Returns

kStatus_Success - Set CAN FD bit rate (include Nominal and Data phase) successfully.

void FLEXCAN_SetRxMbGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive message buffer global mask.

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Message Buffer Global Mask value.

void FLEXCAN_SetRxFifoGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive FIFO global mask.

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Fifo Global Mask value.

void FLEXCAN_SetRxIndividualMask(CAN_Type *base, uint8_t maskIdx, uint32_t mask)

Sets the FlexCAN receive individual mask.

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the FLEXCAN_Init(). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

- base – FlexCAN peripheral base address.
- maskIdx – The Index of individual Mask.
- mask – Rx Individual Mask value.

void FLEXCAN_SetTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

void FLEXCAN_SetRxMbConfig(CAN_Type *base, uint8_t mbIdx, const flexcan_rx_mb_config_t *pRxMbConfig, bool enable)

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer. User should invoke this API when CTRL2[RRS]=1. When CTRL2[RRS]=1, frame's

ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbEmpty, kFLEXCAN_RxMbFull or kFLEXCAN_RxMbOverrun. Message buffer will store the remote frame in the same fashion of a data frame. No automatic remote response frame will be generated. User need to setup another message buffer to respond remote request.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

void FLEXCAN_SetFDTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

void FLEXCAN_SetFDRxMbConfig(CAN_Type *base, uint8_t mbIdx, const *flexcan_rx_mb_config_t* *pRxMbConfig, bool enable)

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

void FLEXCAN_SetRemoteResponseMbConfig(CAN_Type *base, uint8_t mbIdx, const *flexcan_frame_t* *pFrame)

Configures a FlexCAN Remote Response Message Buffer.

User should invoke this API when CTRL2[RRS]=0. When CTRL2[RRS]=0, frame's ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbRanswer. If there is a matching ID, then this mailbox content will be transmitted as response. The received remote request frame is not stored in receive buffer. It is only used to trigger a transmission of a frame in response.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pFrame – Pointer to CAN message frame structure for response.

```
void FLEXCAN_SetRxFifoConfig(CAN_Type *base, const flexcan_rx_fifo_config_t *pRxFifoConfig,  
                             bool enable)
```

Configures the FlexCAN Legacy Rx FIFO.

This function configures the FlexCAN Rx FIFO with given configuration.

Note: Legacy Rx FIFO only can receive classic CAN message.

Parameters

- base – FlexCAN peripheral base address.
- pRxFifoConfig – Pointer to the FlexCAN Legacy Rx FIFO configuration structure. Can be NULL when enable parameter is false.
- enable – Enable/disable Legacy Rx FIFO.
 - true: Enable Legacy Rx FIFO.
 - false: Disable Legacy Rx FIFO.

```
void FLEXCAN_SetEnhancedRxFifoConfig(CAN_Type *base, const  
                                     flexcan_enhanced_rx_fifo_config_t *pConfig, bool  
                                     enable)
```

Configures the FlexCAN Enhanced Rx FIFO.

This function configures the Enhanced Rx FIFO with given configuration.

Note: Enhanced Rx FIFO support receive classic CAN or CAN FD messages, Legacy Rx FIFO and Enhanced Rx FIFO cannot be enabled at the same time.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Enhanced Rx FIFO configuration structure. Can be NULL when enable parameter is false.
- enable – Enable/disable Enhanced Rx FIFO.
 - true: Enable Enhanced Rx FIFO.
 - false: Disable Enhanced Rx FIFO.

```
void FLEXCAN_SetPNConfig(CAN_Type *base, const flexcan_pn_config_t *pConfig)
```

Configures the FlexCAN Pretended Networking mode.

This function configures the FlexCAN Pretended Networking mode with given configuration.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Rx FIFO configuration structure.

```
static inline uint64_t FLEXCAN_GetStatusFlags(CAN_Type *base)
```

Gets the FlexCAN module interrupt flags.

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators `_flexcan_flags`. To check the specific status, compare the return value with enumerators in `_flexcan_flags`.

Parameters

- `base` – FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the `_flexcan_flags`.

```
static inline void FLEXCAN_ClearStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears status flags with the provided mask.

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

- `base` – FlexCAN peripheral base address.
- `mask` – The status flags to be cleared, it is logical OR value of `_flexcan_flags`.

```
static inline void FLEXCAN_GetBusErrCount(CAN_Type *base, uint8_t *txErrBuf, uint8_t  
                                         *rxErrBuf)
```

Gets the FlexCAN Bus Error Counter value.

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

- `base` – FlexCAN peripheral base address.
- `txErrBuf` – Buffer to store Tx Error Counter value.
- `rxErrBuf` – Buffer to store Rx Error Counter value.

```
static inline uint64_t FLEXCAN_GetMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Gets the FlexCAN low 64 Message Buffer interrupt flags.

This function gets the interrupt flags of a given Message Buffers.

Parameters

- `base` – FlexCAN peripheral base address.
- `mask` – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

```
static inline uint64_t FLEXCAN_GetHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)
```

Gets the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- `base` – FlexCAN peripheral base address.
- `mask` – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

```
static inline void FLEXCAN_ClearMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears the FlexCAN low 64 Message Buffer interrupt flags.

This function clears the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_ClearHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
void FLEXCAN_GetMemoryErrorReportStatus(CAN_Type *base,  
                                         flexcan_memory_error_report_status_t  
                                         *errorStatus)
```

Gets the FlexCAN Memory Error Report registers status.

This function gets the FlexCAN Memory Error Report registers status.

Parameters

- base – FlexCAN peripheral base address.
- errorStatus – Pointer to FlexCAN Memory Error Report registers status structure.

```
static inline uint8_t FLEXCAN_GetPNMatchCount(CAN_Type *base)
```

Gets the FlexCAN Number of Matches when in Pretended Networking.

This function gets the number of times a given message has matched the predefined filtering criteria for ID and/or PL before a wakeup event.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of received wake up messages.

```
static inline uint32_t FLEXCAN_GetEnhancedFifoDataCount(CAN_Type *base)
```

Gets the number of FlexCAN Enhanced Rx FIFO available frames.

This function gets the number of CAN messages stored in the Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of available CAN messages stored in the Enhanced Rx FIFO.

```
static inline void FLEXCAN_EnableInterrupts(CAN_Type *base, uint64_t mask)
```

Enables FlexCAN interrupts according to the provided mask.

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.

- mask – The interrupts to enable. Logical OR of `_flexcan_interrupt_enable`.

`static inline void FLEXCAN_DisableInterrupts(CAN_Type *base, uint64_t mask)`

Disables FlexCAN interrupts according to the provided mask.

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.
- mask – The interrupts to disable. Logical OR of `_flexcan_interrupt_enable`.

`static inline void FLEXCAN_EnableMbInterrupts(CAN_Type *base, uint64_t mask)`

Enables FlexCAN low 64 Message Buffer interrupts.

This function enables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

`static inline void FLEXCAN_EnableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)`

Enables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

`static inline void FLEXCAN_DisableMbInterrupts(CAN_Type *base, uint64_t mask)`

Disables FlexCAN low 64 Message Buffer interrupts.

This function disables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

`static inline void FLEXCAN_DisableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)`

Disables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

`void FLEXCAN_EnableRxFifoDMA(CAN_Type *base, bool enable)`

Enables or disables the FlexCAN Rx FIFO DMA request.

This function enables or disables the DMA feature of FlexCAN build-in Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- enable – true to enable, false to disable.

static inline uintptr_t FLEXCAN_GetRxFifoHeadAddr(CAN_Type *base)

Gets the Rx FIFO Head address.

This function returns the FlexCAN Rx FIFO Head address, which is mainly used for the DMA/eDMA use case.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN Rx FIFO Head address.

static inline status_t FLEXCAN_Enable(CAN_Type *base, bool enable)

Enables or disables the FlexCAN module operation.

This function enables or disables the FlexCAN module.

Parameters

- base – FlexCAN base pointer.
- enable – true to enable, false to disable.

Returns

kStatus_Success Enable FlexCAN module successful
kStatus_Timeout Timeout when wait for Low-Power Mode Acknowledge

status_t FLEXCAN_WriteTxMb(CAN_Type *base, uint8_t mbIdx, const flexcan_frame_t *pTxFrame)

Writes a FlexCAN Message to the Transmit Message Buffer.

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success -- Write Tx Message Buffer Successfully.
- kStatus_Fail -- Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadRxMb(CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *pRxFrame)

Reads a FlexCAN Message from Receive Message Buffer.

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success -- Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow -- Rx Message Buffer is already overflowed and has been read successfully.

- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_WriteFDTxMb(CAN_Type *base, uint8_t mbIdx, const *flexcan_fd_frame_t* *pTxFrame)

Writes a FlexCAN FD Message to the Transmit Message Buffer.

This function writes a CAN FD Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN FD Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – Write Tx Message Buffer Successfully.
- kStatus_Fail – Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadFDRxMb(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN FD Message from Receive Message Buffer.

This function reads a CAN FD message from a specified Receive Message Buffer. The function fills a receive CAN FD message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_ReadRxFifo(CAN_Type *base, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Legacy Rx FIFO.

This function reads a CAN message from the FlexCAN Legacy Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.

status_t FLEXCAN_ReadEnhancedRxFifo(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN Message from Enhanced Rx FIFO.

This function reads a CAN or CAN FD message from the FlexCAN Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.

status_t FLEXCAN_ReadPNWakeUpMB(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Wake Up MB.

This function reads a CAN message from the FlexCAN Wake up Message Buffers. There are four Wake up Message Buffers (WMBs) used to store incoming messages in Pretended Networking mode. The WMB index indicates the arrival order. The last message is stored in WMB3.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN message frame structure for reception.
- mbIdx – The FlexCAN Wake up Message Buffer index. Range in 0x0 ~ 0x3.

Return values

- kStatus_Success – - Read Message from Wake up Message Buffer successfully.
- kStatus_Fail – - Wake up Message Buffer has no valid content.

status_t FLEXCAN_TransferFDSendBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pTxFrame)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.
- kStatus_Timeout – - Failed to send frames within specific time.

status_t FLEXCAN_TransferFDReceiveBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.

- `mbIdx` – The FlexCAN FD Message Buffer index.
- `pRxFrame` – Pointer to CAN FD message frame structure for reception.

Return values

- `kStatus_Success` – Rx Message Buffer is full and has been read successfully.
- `kStatus_FLEXCAN_RxOverflow` – Rx Message Buffer is already overflowed and has been read successfully.
- `kStatus_Fail` – Rx Message Buffer is empty.
- `kStatus_Timeout` – Failed to receive frames within specific time.

`status_t FLEXCAN_TransferFDSendNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *pMbXfer)`

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `pMbXfer` – FlexCAN FD Message Buffer transfer structure. See the `flexcan_mb_transfer_t`.

Return values

- `kStatus_Success` – Start Tx Message Buffer sending process successfully.
- `kStatus_Fail` – Write Tx Message Buffer failed.
- `kStatus_FLEXCAN_TxBusy` – Tx Message Buffer is in use.

`status_t FLEXCAN_TransferFDReceiveNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *pMbXfer)`

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `pMbXfer` – FlexCAN FD Message Buffer transfer structure. See the `flexcan_mb_transfer_t`.

Return values

- `kStatus_Success` – Start Rx Message Buffer receiving process successfully.
- `kStatus_FLEXCAN_RxBusy` – Rx Message Buffer is in use.

`void FLEXCAN_TransferFDAbortSend(CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)`

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.

- mbIdx – The FlexCAN FD Message Buffer index.

void FLEXCAN_TransferFDAbortReceive(CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN FD Message Buffer index.

status_t FLEXCAN_TransferSendBlocking(CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *pTxFrame)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.
- kStatus_Timeout – - Failed to send frames within specific time.

status_t FLEXCAN_TransferReceiveBlocking(CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveFifoBlocking(CAN_Type *base, *flexcan_frame_t* *pRxFrame)
 Performs a polling receive transaction from Legacy Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.
- kStatus_Timeout – Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveEnhancedFifoBlocking(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction from Enhanced Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.
- kStatus_Timeout – Failed to receive frames within specific time.

void FLEXCAN_TransferCreateHandle(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_transfer_callback_t* callback, void *userData)

Initializes the FlexCAN handle.

This function initializes the FlexCAN handle, which can be used for other FlexCAN transactional APIs. Usually, for a specified FlexCAN instance, call this API once to get the initialized handle.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

status_t FLEXCAN_TransferSendNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

status_t FLEXCAN_TransferReceiveNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *pMbXfer)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Rx Message Buffer receiving process successfully.
- kStatus_FLEXCAN_RxBusy – Rx Message Buffer is in use.

status_t FLEXCAN_TransferReceiveFifoNonBlocking(CAN_Type *base, flexcan_handle_t *handle, flexcan_fifo_transfer_t *pFifoXfer)

Receives a message from Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pFifoXfer – FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t.

Return values

- kStatus_Success – Start Rx FIFO receiving process successfully.
- kStatus_FLEXCAN_RxFifoBusy – Rx FIFO is currently in use.

status_t FLEXCAN_TransferGetReceiveFifoCount(CAN_Type *base, flexcan_handle_t *handle, size_t *count)

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – count is Invalid.
- `kStatus_Success` – Successfully return the count.

`status_t` FLEXCAN_TransferReceiveEnhancedFifoNonBlocking(`CAN_Type` *base, *flexcan_handle_t* *handle, *flexcan_fifo_transfer_t* *pFifoXfer)

Receives a message from Enhanced Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pFifoXfer – FlexCAN Rx FIFO transfer structure. See the ref *flexcan_fifo_transfer_t*.

Return values

- `kStatus_Success` – Start Rx FIFO receiving process successfully.
- `kStatus_FLEXCAN_RxFifoBusy` – Rx FIFO is currently in use.

static inline `status_t` FLEXCAN_TransferGetReceiveEnhancedFifoCount(`CAN_Type` *base, *flexcan_handle_t* *handle, `size_t` *count)

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – count is Invalid.
- `kStatus_Success` – Successfully return the count.

`uint32_t` FLEXCAN_GetTimeStamp(*flexcan_handle_t* *handle, `uint8_t` mbIdx)

Gets the detail index of Mailbox's Timestamp by handle.

Then function can only be used when calling non-blocking Data transfer (TX/RX) API, After TX/RX data transfer done (User can get the status by handler's callback function), we can get the detail index of Mailbox's timestamp by handle, Detail non-blocking data transfer API (TX/RX) contain. -FLEXCAN_TransferSendNonBlocking -FLEXCAN_TransferFDSendNonBlocking -FLEXCAN_TransferReceiveNonBlocking -FLEXCAN_TransferFDReceiveNonBlocking -FLEXCAN_TransferReceiveFifoNonBlocking

Parameters

- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

Return values

the – index of mailbox 's timestamp stored in the handle.

void FLEXCAN_TransferAbortSend(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

void FLEXCAN_TransferAbortReceive(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

void FLEXCAN_TransferAbortReceiveFifo(CAN_Type *base, *flexcan_handle_t* *handle)

Aborts the interrupt driven message receive from Rx FIFO process.

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_TransferAbortReceiveEnhancedFifo(CAN_Type *base, *flexcan_handle_t* *handle)

Aborts the interrupt driven message receive from Enhanced Rx FIFO process.

This function aborts the interrupt driven message receive from Enhanced Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_TransferHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN IRQ handle function.

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_MbHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle, uint32_t startMbIdx, uint32_t endMbIdx)

FlexCAN Message Buffer IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- startMbIdx – First Message Buffer to handle.
- endMbIdx – Last Message Buffer to handle.

void FLEXCAN_EhancedRxFifoHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)
FlexCAN Enhanced Rx FIFO IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_BusoffErrorHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)
FlexCAN Bus Off, Error and Warning IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_PNWakeUpHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)
FlexCAN Pretended Networking Wake-up IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_MemoryErrorHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)
FlexCAN Memory Error IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

FSL_FLEXCAN_DRIVER_VERSION
FlexCAN driver version.

FlexCAN transfer status.

Values:

enumerator kStatus_FLEXCAN_TxBusy
Tx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_TxIdle
Tx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_TxSwitchToRx
Remote Message is send out and Message buffer changed to Receive one.

enumerator kStatus_FLEXCAN_RxBusy
Rx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_RxIdle
Rx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_RxOverflow
Rx Message Buffer is Overflowed.

enumerator kStatus_FLEXCAN_RxFifoBusy
Rx Message FIFO is Busy.

enumerator kStatus_FLEXCAN_RxFifoIdle
Rx Message FIFO is Idle.

enumerator kStatus_FLEXCAN_RxFifoOverflow
Rx Message FIFO is overflowed.

enumerator kStatus_FLEXCAN_RxFifoWarning
Rx Message FIFO is almost overflowed.

enumerator kStatus_FLEXCAN_RxFifoDisabled
Rx Message FIFO is disabled during reading.

enumerator kStatus_FLEXCAN_ErrorStatus
FlexCAN Module Error and Status.

enumerator kStatus_FLEXCAN_WakeUp
FlexCAN is waken up from STOP mode.

enumerator kStatus_FLEXCAN_UnHandled
UnHandled Interrupt asserted.

enumerator kStatus_FLEXCAN_RxRemote
Rx Remote Message Received in Mail box.

enumerator kStatus_FLEXCAN_RxFifoUnderflow
Enhanced Rx Message FIFO is underflow.

enumerator kStatus_FLEXCAN_MemoryError
FlexCAN Memory Error.

enum _flexcan_frame_format
FlexCAN frame format.

Values:

enumerator kFLEXCAN_FrameFormatStandard
Standard frame format attribute.

enumerator kFLEXCAN_FrameFormatExtend
Extend frame format attribute.

enum _flexcan_frame_type
FlexCAN frame type.

Values:

enumerator kFLEXCAN_FrameTypeData
Data frame type attribute.

enumerator kFLEXCAN_FrameTypeRemote
Remote frame type attribute.

enum _flexcan_clock_source
FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superseded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superseded kFLEXCAN_ClkSrc1

Values:

enumerator kFLEXCAN_ClkSrcOsc
FlexCAN Protocol Engine clock from Oscillator.

enumerator kFLEXCAN_ClkSrcPeri

FlexCAN Protocol Engine clock from Peripheral Clock.

enumerator kFLEXCAN_ClkSrc0

FlexCAN Protocol Engine clock selected by user as SRC == 0.

enumerator kFLEXCAN_ClkSrc1

FlexCAN Protocol Engine clock selected by user as SRC == 1.

enum _flexcan_wake_up_source

FlexCAN wake up source.

Values:

enumerator kFLEXCAN_WakeupSrcUnfiltered

FlexCAN uses unfiltered Rx input to detect edge.

enumerator kFLEXCAN_WakeupSrcFiltered

FlexCAN uses filtered Rx input to detect edge.

enum _flexcan_rx_fifo_filter_type

FlexCAN Rx Fifo Filter type.

Values:

enumerator kFLEXCAN_RxFifoFilterTypeA

One full ID (standard and extended) per ID Filter element.

enumerator kFLEXCAN_RxFifoFilterTypeB

Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

enumerator kFLEXCAN_RxFifoFilterTypeC

Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

enumerator kFLEXCAN_RxFifoFilterTypeD

All frames rejected.

enum _flexcan_mb_size

FlexCAN Message Buffer Payload size.

Values:

enumerator kFLEXCAN_8BperMB

Selects 8 bytes per Message Buffer.

enumerator kFLEXCAN_16BperMB

Selects 16 bytes per Message Buffer.

enumerator kFLEXCAN_32BperMB

Selects 32 bytes per Message Buffer.

enumerator kFLEXCAN_64BperMB

Selects 64 bytes per Message Buffer.

enum _flexcan_fd_frame_length

FlexCAN CAN FD frame supporting data length (available DLC values).

For Tx, when the Data size corresponding to DLC value stored in the MB selected for transmission is larger than the MB Payload size, FlexCAN adds the necessary number of bytes with constant 0xCC pattern to complete the expected DLC. For Rx, when the Data size corresponding to DLC value received from the CAN bus is larger than the MB Payload size, the high order bytes that do not fit the Payload size will lose.

Values:

enumerator kFLEXCAN_0BperFrame
Frame contains 0 valid data bytes.

enumerator kFLEXCAN_1BperFrame
Frame contains 1 valid data bytes.

enumerator kFLEXCAN_2BperFrame
Frame contains 2 valid data bytes.

enumerator kFLEXCAN_3BperFrame
Frame contains 3 valid data bytes.

enumerator kFLEXCAN_4BperFrame
Frame contains 4 valid data bytes.

enumerator kFLEXCAN_5BperFrame
Frame contains 5 valid data bytes.

enumerator kFLEXCAN_6BperFrame
Frame contains 6 valid data bytes.

enumerator kFLEXCAN_7BperFrame
Frame contains 7 valid data bytes.

enumerator kFLEXCAN_8BperFrame
Frame contains 8 valid data bytes.

enumerator kFLEXCAN_12BperFrame
Frame contains 12 valid data bytes.

enumerator kFLEXCAN_16BperFrame
Frame contains 16 valid data bytes.

enumerator kFLEXCAN_20BperFrame
Frame contains 20 valid data bytes.

enumerator kFLEXCAN_24BperFrame
Frame contains 24 valid data bytes.

enumerator kFLEXCAN_32BperFrame
Frame contains 32 valid data bytes.

enumerator kFLEXCAN_48BperFrame
Frame contains 48 valid data bytes.

enumerator kFLEXCAN_64BperFrame
Frame contains 64 valid data bytes.

enum _flexcan_efifo_dma_per_read_length
FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

Values:

enumerator kFLEXCAN_1WordPerRead
Transfer 1 32-bit words (CS).

enumerator kFLEXCAN_2WordPerRead
Transfer 2 32-bit words (CS + ID).

enumerator kFLEXCAN_3WordPerRead
Transfer 3 32-bit words (CS + ID + 1~4 bytes data).

enumerator kFLEXCAN_4WordPerRead

Transfer 4 32-bit words (CS + ID + 5~8 bytes data).

enumerator kFLEXCAN_5WordPerRead

Transfer 5 32-bit words (CS + ID + 9~12 bytes data).

enumerator kFLEXCAN_6WordPerRead

Transfer 6 32-bit words (CS + ID + 13~16 bytes data).

enumerator kFLEXCAN_7WordPerRead

Transfer 7 32-bit words (CS + ID + 17~20 bytes data).

enumerator kFLEXCAN_8WordPerRead

Transfer 8 32-bit words (CS + ID + 21~24 bytes data).

enumerator kFLEXCAN_9WordPerRead

Transfer 9 32-bit words (CS + ID + 25~28 bytes data).

enumerator kFLEXCAN_10WordPerRead

Transfer 10 32-bit words (CS + ID + 29~32 bytes data).

enumerator kFLEXCAN_11WordPerRead

Transfer 11 32-bit words (CS + ID + 33~36 bytes data).

enumerator kFLEXCAN_12WordPerRead

Transfer 12 32-bit words (CS + ID + 37~40 bytes data).

enumerator kFLEXCAN_13WordPerRead

Transfer 13 32-bit words (CS + ID + 41~44 bytes data).

enumerator kFLEXCAN_14WordPerRead

Transfer 14 32-bit words (CS + ID + 45~48 bytes data).

enumerator kFLEXCAN_15WordPerRead

Transfer 15 32-bit words (CS + ID + 49~52 bytes data).

enumerator kFLEXCAN_16WordPerRead

Transfer 16 32-bit words (CS + ID + 53~56 bytes data).

enumerator kFLEXCAN_17WordPerRead

Transfer 17 32-bit words (CS + ID + 57~60 bytes data).

enumerator kFLEXCAN_18WordPerRead

Transfer 18 32-bit words (CS + ID + 61~64 bytes data).

enumerator kFLEXCAN_19WordPerRead

Transfer 19 32-bit words (CS + ID + 64 bytes data + ID HIT).

enum _flexcan_rx_fifo_priority

FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

Values:

enumerator kFLEXCAN_RxFifoPrioLow

Matching process start from Rx Message Buffer first.

enumerator kFLEXCAN_RxFifoPrioHigh

Matching process start from Enhanced/Legacy Rx FIFO first.

enum _flexcan_interrupt_enable

FlexCAN interrupt enable enumerations.

This provides constants for the FlexCAN interrupt enable enumerations for use in the FlexCAN functions.

Note: FlexCAN Message Buffers and Legacy Rx FIFO interrupts not included in.

Values:

enumerator kFLEXCAN_BusOffInterruptEnable

Bus Off interrupt, use bit 15.

enumerator kFLEXCAN_ErrorInterruptEnable

CAN Error interrupt, use bit 14.

enumerator kFLEXCAN_TxWarningInterruptEnable

Tx Warning interrupt, use bit 11.

enumerator kFLEXCAN_RxWarningInterruptEnable

Rx Warning interrupt, use bit 10.

enumerator kFLEXCAN_WakeUpInterruptEnable

Self Wake Up interrupt, use bit 26.

enumerator kFLEXCAN_FDErrorInterruptEnable

CAN FD Error interrupt, use bit 31.

enumerator kFLEXCAN_PNMatchWakeUpInterruptEnable

PN Match Wake Up interrupt, use high word bit 17.

enumerator kFLEXCAN_PNTimeoutWakeUpInterruptEnable

PN Timeout Wake Up interrupt, use high word bit 16. Enhanced Rx FIFO Underflow interrupt, use high word bit 31.

enumerator kFLEXCAN_ERxFifoUnderflowInterruptEnable

Enhanced Rx FIFO Overflow interrupt, use high word bit 30.

enumerator kFLEXCAN_ERxFifoOverflowInterruptEnable

Enhanced Rx FIFO Watermark interrupt, use high word bit 29.

enumerator kFLEXCAN_ERxFifoWatermarkInterruptEnable

Enhanced Rx FIFO Data Available interrupt, use high word bit 28.

enumerator kFLEXCAN_ERxFifoDataAvlInterruptEnable

enumerator kFLEXCAN_HostAccessNCErrInterruptEnable

Host Access With Non-Correctable Errors interrupt, use high word bit 0.

enumerator kFLEXCAN_FlexCanAccessNCErrInterruptEnable

FlexCAN Access With Non-Correctable Errors interrupt, use high word bit 2.

enumerator kFLEXCAN_HostOrFlexCanCErrInterruptEnable

Host or FlexCAN Access With Correctable Errors interrupt, use high word bit 3.

enum _flexcan_flags

FlexCAN status flags.

This provides constants for the FlexCAN status flags for use in the FlexCAN functions.

Note: The CPU read action clears the bits corresponding to the FLEXCAN_ErrorFlag macro, therefore user need to read status flags and distinguish which error is occur using _flexcan_error_flags enumerations.

Values:

enumerator kFLEXCAN_ErrorOverrunFlag
Error Overrun Status.

enumerator kFLEXCAN_FDErrorIntFlag
CAN FD Error Interrupt Flag.

enumerator kFLEXCAN_BusoffDoneIntFlag
Bus Off process completed Interrupt Flag.

enumerator kFLEXCAN_SynchFlag
CAN Synchronization Status.

enumerator kFLEXCAN_TxWarningIntFlag
Tx Warning Interrupt Flag.

enumerator kFLEXCAN_RxWarningIntFlag
Rx Warning Interrupt Flag.

enumerator kFLEXCAN_IdleFlag
FlexCAN In IDLE Status.

enumerator kFLEXCAN_FaultConfinementFlag
FlexCAN Fault Confinement State.

enumerator kFLEXCAN_TransmittingFlag
FlexCAN In Transmission Status.

enumerator kFLEXCAN_ReceivingFlag
FlexCAN In Reception Status.

enumerator kFLEXCAN_BusOffIntFlag
Bus Off Interrupt Flag.

enumerator kFLEXCAN_ErrorIntFlag
CAN Error Interrupt Flag.

enumerator kFLEXCAN_WakeUpIntFlag
Self Wake-Up Interrupt Flag.

enumerator kFLEXCAN_ErrorFlag

enumerator kFLEXCAN_PNMatchIntFlag
PN Matching Event Interrupt Flag.

enumerator kFLEXCAN_PNTimeoutIntFlag
PN Timeout Event Interrupt Flag.

enumerator kFLEXCAN_ERxFifoUnderflowIntFlag
Enhanced Rx FIFO underflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoOverflowIntFlag
Enhanced Rx FIFO overflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoWatermarkIntFlag
Enhanced Rx FIFO watermark Interrupt Flag.

enumerator kFLEXCAN_ERxFifoDataAvlIntFlag

Enhanced Rx FIFO data available Interrupt Flag.

enumerator kFLEXCAN_ERxFifoEmptyFlag

Enhanced Rx FIFO empty status.

enumerator kFLEXCAN_ERxFifoFullFlag

Enhanced Rx FIFO full status.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorIntFlag

Host Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorIntFlag

FlexCAN Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_CorrectableErrorIntFlag

Correctable Error Interrupt Flag.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorOverrunFlag

Host Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorOverrunFlag

FlexCAN Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_CorrectableErrorOverrunFlag

Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_AllMemoryErrorIntFlag

All Memory Error Interrupt Flags.

enumerator kFLEXCAN_AllMemoryErrorFlag

All Memory Error Flags.

enum _flexcan_error_flags

FlexCAN error status flags.

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN_ErrorFlag in _flexcan_flags enumerations to determine which error is generated.

Values:

enumerator kFLEXCAN_FDStuffingError

Stuffing Error.

enumerator kFLEXCAN_FDFormError

Form Error.

enumerator kFLEXCAN_FDCrcError

Cyclic Redundancy Check Error.

enumerator kFLEXCAN_FDBit0Error

Unable to send dominant bit.

enumerator kFLEXCAN_FDBit1Error

Unable to send recessive bit.

enumerator kFLEXCAN_TxErrorWarningFlag

Tx Error Warning Status.

enumerator kFLEXCAN_RxErrorWarningFlag

Rx Error Warning Status.

enumerator kFLEXCAN_StuffingError

Stuffing Error.

enumerator kFLEXCAN_FormError

Form Error.

enumerator kFLEXCAN_CrcError

Cyclic Redundancy Check Error.

enumerator kFLEXCAN_AckError

Received no ACK on transmission.

enumerator kFLEXCAN_Bit0Error

Unable to send dominant bit.

enumerator kFLEXCAN_Bit1Error

Unable to send recessive bit.

FlexCAN Legacy Rx FIFO status flags.

The FlexCAN Legacy Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupiees more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Values:

enumerator kFLEXCAN_RxFifoOverflowFlag

Rx FIFO overflow flag.

enumerator kFLEXCAN_RxFifoWarningFlag

Rx FIFO almost full flag.

enumerator kFLEXCAN_RxFifoFrameAvlFlag

Frames available in Rx FIFO flag.

enum _flexcan_memory_error_type

FlexCAN Memory Error Type.

Values:

enumerator kFLEXCAN_CorrectableError

The memory error is correctable which means on bit error.

enumerator kFLEXCAN_NonCorrectableError

The memory error is non-correctable which means two bit errors.

enum _flexcan_memory_access_type

FlexCAN Memory Access Type.

Values:

enumerator kFLEXCAN_MoveOutFlexCanAccess

The memory error was detected during move-out FlexCAN access.

enumerator kFLEXCAN_MoveInAccess

The memory error was detected during move-in FlexCAN access.

enumerator kFLEXCAN_TxArbitrationAccess

The memory error was detected during Tx Arbitration FlexCAN access.

enumerator kFLEXCAN_RxMatchingAccess

The memory error was detected during Rx Matching FlexCAN access.

enumerator kFLEXCAN_MoveOutHostAccess

The memory error was detected during Rx Matching Host (CPU) access.

enum _flexcan_byte_error_syndrome

FlexCAN Memory Error Byte Syndrome.

Values:

enumerator kFLEXCAN_NoError

No bit error in this byte.

enumerator kFLEXCAN_ParityBits0Error

Parity bit 0 error in this byte.

enumerator kFLEXCAN_ParityBits1Error

Parity bit 1 error in this byte.

enumerator kFLEXCAN_ParityBits2Error

Parity bit 2 error in this byte.

enumerator kFLEXCAN_ParityBits3Error

Parity bit 3 error in this byte.

enumerator kFLEXCAN_ParityBits4Error

Parity bit 4 error in this byte.

enumerator kFLEXCAN_DataBits0Error

Data bit 0 error in this byte.

enumerator kFLEXCAN_DataBits1Error

Data bit 1 error in this byte.

enumerator kFLEXCAN_DataBits2Error

Data bit 2 error in this byte.

enumerator kFLEXCAN_DataBits3Error

Data bit 3 error in this byte.

enumerator kFLEXCAN_DataBits4Error

Data bit 4 error in this byte.

enumerator kFLEXCAN_DataBits5Error

Data bit 5 error in this byte.

enumerator kFLEXCAN_DataBits6Error

Data bit 6 error in this byte.

enumerator kFLEXCAN_DataBits7Error

Data bit 7 error in this byte.

enumerator kFLEXCAN_AllZeroError

All-zeros non-correctable error in this byte.

enumerator kFLEXCAN_AllOneError

All-ones non-correctable error in this byte.

enumerator kFLEXCAN_NonCorrectableErrors

Non-correctable error in this byte.

enum _flexcan_pn_match_source

FlexCAN Pretended Networking match source selection.

Values:

enumerator kFLEXCAN_PNMatSrcID

Message match with ID filtering.

enumerator kFLEXCAN_PNMatSrcIDAndData

Message match with ID filtering and payload filtering.

enum *_flexcan_pn_match_mode*

FlexCAN Pretended Networking mode match type.

Values:

enumerator kFLEXCAN_PNMatModeEqual

Match upon ID/Payload contents against an exact target value.

enumerator kFLEXCAN_PNMatModeGreater

Match upon an ID/Payload value greater than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeSmaller

Match upon an ID/Payload value smaller than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeRange

Match upon an ID/Payload value inside a range, greater than or equal to a specified lower limit, and smaller than or equal to a specified upper limit

typedef enum *_flexcan_frame_format* flexcan_frame_format_t

FlexCAN frame format.

typedef enum *_flexcan_frame_type* flexcan_frame_type_t

FlexCAN frame type.

typedef enum *_flexcan_clock_source* flexcan_clock_source_t

FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

typedef enum *_flexcan_wake_up_source* flexcan_wake_up_source_t

FlexCAN wake up source.

typedef enum *_flexcan_rx_fifo_filter_type* flexcan_rx_fifo_filter_type_t

FlexCAN Rx Fifo Filter type.

typedef enum *_flexcan_mb_size* flexcan_mb_size_t

FlexCAN Message Buffer Payload size.

typedef enum *_flexcan_efifo_dma_per_read_length* flexcan_efifo_dma_per_read_length_t

FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

typedef enum *_flexcan_rx_fifo_priority* flexcan_rx_fifo_priority_t

FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

typedef enum *_flexcan_memory_error_type* flexcan_memory_error_type_t

FlexCAN Memory Error Type.

typedef enum *_flexcan_memory_access_type* flexcan_memory_access_type_t

FlexCAN Memory Access Type.

typedef enum *_flexcan_byte_error_syndrome* flexcan_byte_error_syndrome_t

FlexCAN Memory Error Byte Syndrome.

typedef struct *_flexcan_memory_error_report_status* flexcan_memory_error_report_status_t

FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of FLEXCAN_GetMemoryErrorReportStatus() function. And user can use FLEXCAN_GetMemoryErrorReportStatus to get the status of the last memory error access.

typedef struct *_flexcan_frame* flexcan_frame_t

FlexCAN message frame structure.

typedef struct *_flexcan_fd_frame* flexcan_fd_frame_t

CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see *_flexcan_fd_frame_length*.

typedef struct *_flexcan_timing_config* flexcan_timing_config_t

FlexCAN protocol timing characteristic configuration structure.

typedef struct *_flexcan_config* flexcan_config_t

FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

typedef struct *_flexcan_rx_mb_config* flexcan_rx_mb_config_t

FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

typedef enum *_flexcan_pn_match_source* flexcan_pn_match_source_t

FlexCAN Pretended Networking match source selection.

typedef enum *_flexcan_pn_match_mode* flexcan_pn_match_mode_t

FlexCAN Pretended Networking mode match type.

typedef struct *_flexcan_pn_config* flexcan_pn_config_t

FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of FLEXCAN_SetPNConfig() function. The FLEXCAN_SetPNConfig() function is used to configure FlexCAN Networking work mode.

typedef struct *_flexcan_rx_fifo_config* flexcan_rx_fifo_config_t

FlexCAN Legacy Rx FIFO configuration structure.

typedef struct *_flexcan_enhanced_rx_fifo_std_id_filter* flexcan_enhanced_rx_fifo_std_id_filter_t

FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

typedef struct *_flexcan_enhanced_rx_fifo_ext_id_filter* flexcan_enhanced_rx_fifo_ext_id_filter_t

FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

`typedef struct flexcan_enhanced_rx_fifo_config flexcan_enhanced_rx_fifo_config_t`
 FlexCAN Enhanced Rx FIFO configuration structure.

`typedef struct flexcan_mb_transfer flexcan_mb_transfer_t`
 FlexCAN Message Buffer transfer.

`typedef struct flexcan_fifo_transfer flexcan_fifo_transfer_t`
 FlexCAN Rx FIFO transfer.

`typedef struct flexcan_handle flexcan_handle_t`
 FlexCAN handle structure definition.

`typedef void (*flexcan_transfer_callback_t)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint64_t result, void *userData)`

`FLEXCAN_WAIT_TIMEOUT`

`FLEXCAN_POLLING_TIMEOUT`
 Max loops to wait for polling transfer.

`FLEXCAN_MODULE_TIMEOUT`
 Max loops to wait for FlexCAN register access complete.

`DLC_LENGTH_DECODE(dlc)`
 FlexCAN frame length helper macro.

`FLEXCAN_ID_STD(id)`
 FlexCAN Frame ID helper macro.
 Standard Frame ID helper macro.

`FLEXCAN_ID_EXT(id)`
 Extend Frame ID helper macro.

`FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)`
 FlexCAN Rx Message Buffer Mask helper macro.
 Standard Rx Message Buffer Mask helper macro.

`FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)`
 Extend Rx Message Buffer Mask helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)`
 FlexCAN Legacy Rx FIFO Mask helper macro.
 Standard Rx FIFO Mask helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)`
 Standard Rx FIFO Mask helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)`
 Standard Rx FIFO Mask helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id)`
 Standard Rx FIFO Mask helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id)`
 Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id)`
 Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)`
 Standard Rx FIFO Mask helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type A helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)

Extend Rx FIFO Mask helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)

Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)

Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)

Extend Rx FIFO Mask helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide)

FlexCAN Rx FIFO Filter helper macro.

Standard Rx FIFO Filter helper macro Type A helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)

Standard Rx FIFO Filter helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)

Standard Rx FIFO Filter helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)

Standard Rx FIFO Filter helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)

Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)

Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id)

Standard Rx FIFO Filter helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide)

Extend Rx FIFO Filter helper macro Type A helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)

Extend Rx FIFO Filter helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)

Extend Rx FIFO Filter helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id)

Extend Rx FIFO Filter helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)

Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)

Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id)

Extend Rx FIFO Filter helper macro Type C lower part helper macro.

ENHANCED_RX_FIFO_FSC(x)

FlexCAN Enhanced Rx FIFO Filter and Mask helper macro.

RTR_STD_HIGH(x)

RTR_STD_LOW(x)

RTR_EXT(x)

ID_STD_LOW(id)

ID_STD_HIGH(id)

ID_EXT(id)

FLEXCAN_ENHANCED_RX_FIFO_STD_MASK_AND_FILTER(id, rtr, id_mask, rtr_mask)

Standard ID filter element with filter + mask scheme.

FLEXCAN_ENHANCED_RX_FIFO_STD_FILTER_WITH_RANGE(id_upper, rtr, id_lower,
rtr_mask)

Standard ID filter element with filter range.

FLEXCAN_ENHANCED_RX_FIFO_STD_TWO_FILTERS(id1, rtr1, id2, rtr2)

Standard ID filter element with two filters without masks.

FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_LOW(id, rtr)

Extended ID filter element with filter + mask scheme low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_HIGH(id_mask, rtr_mask)

Extended ID filter element with filter + mask scheme high word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_LOW(id_upper, rtr)

Extended ID filter element with range scheme low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_HIGH(id_lower, rtr_mask)

Extended ID filter element with range scheme high word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_LOW(id2, rtr2)

Extended ID filter element with two filters without masks low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_HIGH(id1, rtr1)

Extended ID filter element with two filters without masks high word.

FLEXCAN_PN_STD_MASK(id, rtr)

FlexCAN Pretended Networking ID Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

FLEXCAN_PN_EXT_MASK(id, rtr)

Extend Rx Message Buffer Mask helper macro.

FLEXCAN_PN_INT_MASK(x)

FlexCAN interrupt/status flag helper macro.

FLEXCAN_PN_INT_UNMASK(x)

FLEXCAN_PN_STATUS_MASK(x)

FLEXCAN_PN_STATUS_UNMASK(x)

FLEXCAN_EFIFO_INT_MASK(x)

FLEXCAN_EFIFO_INT_UNMASK(x)

FLEXCAN_EFIFO_STATUS_MASK(x)

FLEXCAN_EFIFO_STATUS_UNMASK(x)

FLEXCAN_MECR_INT_MASK(x)

FLEXCAN_MECR_INT_UNMASK(x)

FLEXCAN_MECR_STATUS_MASK(x)

FLEXCAN_MECR_STATUS_UNMASK(x)

FLEXCAN_ERROR_AND_STATUS_INT_FLAG

FLEXCAN_PNWAKE_UP_FLAG

FLEXCAN_WAKE_UP_FLAG

FLEXCAN_MEMORY_ERROR_INT_FLAG

FLEXCAN_MEMORY_ENHANCED_RX_FIFO_INT_FLAG

E_RX_FIFO(base)

FlexCAN Enhanced Rx FIFO base address helper macro.

FLEXCAN_CALLBACK(x)

FlexCAN transfer callback function.

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to `kStatus_FLEXCAN_ErrorStatus`, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

struct flexcan_memory_error_report_status

#include <fsl_flexcan.h> FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of `FLEXCAN_GetMemoryErrorReportStatus()` function. And user can use `FLEXCAN_GetMemoryErrorReportStatus` to get the status of the last memory error access.

Public Members

flexcan_memory_error_type_t errorType

The type of memory error that giving rise to the report.

flexcan_memory_access_type_t accessType

The type of memory access that giving rise to the memory error.

uint16_t accessAddress

The address where memory error detected.

uint32_t errorData

The raw data word read from memory with error.

struct flexcan_frame

#include <fsl_flexcan.h> FlexCAN message frame structure.

struct __flexcan_fd_frame

#include <fsl_flexcan.h> CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see __flexcan_fd_frame_length.

Public Members

uint32_t idhit

Note: ID HIT offset is changed dynamically according to data length code (DLC), when DLC is 15, they will be located below. Using FLEXCAN_FixEnhancedRxFifoFrameIdHit API is recommended to ensure this idhit value is correct. CAN Enhanced Rx FIFO filter hit id (This value is only used in Enhanced Rx FIFO receive mode).

struct __flexcan_timing_config

#include <fsl_flexcan.h> FlexCAN protocol timing characteristic configuration structure.

Public Members

uint16_t preDivider

Classic CAN or CAN FD nominal phase bit rate prescaler.

uint8_t rJumpwidth

Classic CAN or CAN FD nominal phase Re-sync Jump Width.

uint8_t phaseSeg1

Classic CAN or CAN FD nominal phase Segment 1.

uint8_t phaseSeg2

Classic CAN or CAN FD nominal phase Segment 2.

uint8_t propSeg

Classic CAN or CAN FD nominal phase Propagation Segment.

uint16_t fpreDivider

CAN FD data phase bit rate prescaler.

uint8_t frJumpwidth

CAN FD data phase Re-sync Jump Width.

uint8_t fphaseSeg1

CAN FD data phase Phase Segment 1.

uint8_t fphaseSeg2

CAN FD data phase Phase Segment 2.

uint8_t fpropSeg

CAN FD data phase Propagation Segment.

struct __flexcan_config

#include <fsl_flexcan.h> FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

Public Members

flexcan_clock_source_t clkSrc

Clock source for FlexCAN Protocol Engine.

flexcan_wake_up_source_t wakeupSrc

Wake up source selection.

uint8_t maxMbNum

The maximum number of Message Buffers used by user.

bool enableLoopBack

Enable or Disable Loop Back Self Test Mode.

bool enableTimerSync

Enable or Disable Timer Synchronization.

bool enableSelfWakeup

Enable or Disable Self Wakeup Mode.

bool enableIndividMask

Enable or Disable Rx Individual Mask and Queue feature.

bool disableSelfReception

Enable or Disable Self Reflection.

bool enableListenOnlyMode

Enable or Disable Listen Only Mode.

bool enableDoze

Enable or Disable Doze Mode.

bool enablePretendedNetworking

Enable or Disable the Pretended Networking mode.

bool enableMemoryErrorControl

Enable or Disable the memory errors detection and correction mechanism.

bool enableNonCorrectableErrorEnterFreeze

Enable or Disable Non-Correctable Errors In FlexCAN Access Put Device In Freeze Mode.

bool enableTransceiverDelayMeasure

Enable or Disable the transceiver delay measurement, when it is enabled, then the secondary sample point position is determined by the sum of the transceiver delay measurement plus the enhanced TDC offset.

bool enableRemoteRequestFrameStored

true: Store Remote Request Frame in the same fashion of data frame. false: Generate an automatic Remote Response Frame.

struct _flexcan_rx_mb_config

#include <fsl_flexcan.h> FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Public Members

uint32_t id

CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

flexcan_frame_format_t format

CAN Frame Identifier format(Standard of Extend).

flexcan_frame_type_t type

CAN Frame Type(Data or Remote for classical CAN only).

struct *_flexcan_pn_config*

#include <fsl_flexcan.h> FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of FLEXCAN_SetPNConfig() function. The FLEXCAN_SetPNConfig() function is used to configure FlexCAN Networking work mode.

Public Members

bool enableTimeout

Enable or Disable timeout event trigger wakeup.

uint16_t timeoutValue

The timeout value that generates a wakeup event, the counter timer is incremented based on 64 times the CAN Bit Time unit.

bool enableMatch

Enable or Disable match event trigger wakeup.

flexcan_pn_match_source_t matchSrc

Selects the match source (ID and/or data match) to trigger wakeup.

uint8_t matchNum

The number of times a given message must match the predefined ID and/or data before generating a wakeup event, range in 0x1 ~ 0xFF.

flexcan_pn_match_mode_t idMatchMode

The ID match type.

flexcan_pn_match_mode_t dataMatchMode

The data match type.

uint32_t idLower

The ID target values 1 which used either for ID match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in ID match “range detection”.

uint32_t idUpper

The ID target values 2 which used only as the upper limit value in ID match “range detection” or used to store the ID mask in “equal to”.

uint8_t lengthLower

The lower limit for length of data bytes which used only in data match “range detection”. Range in 0x0 ~ 0x8.

uint8_t lengthUpper

The upper limit for length of data bytes which used only in data match “range detection”. Range in 0x0 ~ 0x8.

struct `_flexcan_rx_fifo_config`

#include <fsl_flexcan.h> FlexCAN Legacy Rx FIFO configuration structure.

Public Members

uint32_t *idFilterTable

Pointer to the FlexCAN Legacy Rx FIFO identifier filter table.

uint8_t idFilterNum

The FlexCAN Legacy Rx FIFO Filter elements quantity.

flexcan_rx_fifo_filter_type_t idFilterType

The FlexCAN Legacy Rx FIFO Filter type.

flexcan_rx_fifo_priority_t priority

The FlexCAN Legacy Rx FIFO receive priority.

struct `_flexcan_enhanced_rx_fifo_std_id_filter`

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

Public Members

uint32_t filterType

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1

CAN FD frame data length code (DLC), range see `_flexcan_fd_frame_length`, When the length <= 8, it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use `DLC_LENGTH_DECODE(length)` macro to get the number of valid data bytes.

uint32_t std1

CAN Frame Type(DATA or REMOTE).

uint32_t rtr2

CAN Frame Identifier(STD or EXT format).

uint32_t std2

Substitute Remote request.

struct `_flexcan_enhanced_rx_fifo_ext_id_filter`

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

Public Members

uint32_t filterType

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1

CAN FD frame data length code (DLC), range see `_flexcan_fd_frame_length`, When the length <= 8, it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use `DLC_LENGTH_DECODE(length)` macro to get the number of valid data bytes.

uint32_t std1

CAN Frame Type(DATA or REMOTE).

uint32_t rtr2

CAN Frame Identifier(STD or EXT format).

uint32_t std2

Substitute Remote request.

struct `_flexcan_enhanced_rx_fifo_config`

`#include <fsl_flexcan.h>` FlexCAN Enhanced Rx FIFO configuration structure.

Public Members

uint32_t *idFilterTable

Pointer to the FlexCAN Enhanced Rx FIFO identifier filter table, each table member occupies 32 bit word, table size should be equal to idFilterNum. There are two types of Enhanced Rx FIFO filter elements that can be stored in table : extended-ID filter element (1 word, occupies 1 table members) and standard-ID filter element (2 words, occupies 2 table members), the extended-ID filter element needs to be placed in front of the table.

uint8_t idFilterPairNum

idFilterPairNum is the Enhanced Rx FIFO identifier filter element pair numbers, each pair of filter elements occupies 2 words and can consist of one extended ID filter element or two standard ID filter elements.

uint8_t extendIdFilterNum

The number of extended ID filter element items in the FlexCAN enhanced Rx FIFO identifier filter table, each extended-ID filter element occupies 2 words, extendIdFilterNum need less than or equal to idFilterPairNum.

uint8_t fifoWatermark

(fifoWatermark + 1) is the minimum number of CAN messages stored in the Enhanced RX FIFO which can trigger FIFO watermark interrupt or a DMA request.

flexcan_efifo_dma_per_read_length_t dmaPerReadLength

Define the length of each read of the Enhanced RX FIFO element by the DAM, see `_flexcan_fd_frame_length`.

flexcan_rx_fifo_priority_t priority

The FlexCAN Enhanced Rx FIFO receive priority.

struct `_flexcan_mb_transfer`

`#include <fsl_flexcan.h>` FlexCAN Message Buffer transfer.

Public Members

flexcan_frame_t *frame

The buffer of CAN Message to be transfer.

uint8_t mbIdx

The index of Message buffer used to transfer Message.

struct `_flexcan_fifo_transfer`

`#include <fsl_flexcan.h>` FlexCAN Rx FIFO transfer.

Public Members

flexcan_fd_frame_t *framefd

The buffer of CAN Message to be received from Enhanced Rx FIFO.

flexcan_frame_t *frame

The buffer of CAN Message to be received from Legacy Rx FIFO.

size_t frameNum

Number of CAN Message need to be received from Legacy or Enhanced Rx FIFO.

struct *_flexcan_handle*

#include <fsl_flexcan.h> FlexCAN handle structure.

Public Members

flexcan_transfer_callback_t callback

Callback function.

void *userData

FlexCAN callback function parameter.

flexcan_frame_t *volatile mbFrameBuf[CAN_WORD1_COUNT]

The buffer for received CAN data from Message Buffers.

flexcan_fd_frame_t *volatile mbFDFrameBuf[CAN_WORD1_COUNT]

The buffer for received CAN FD data from Message Buffers.

flexcan_frame_t *volatile rxFifoFrameBuf

The buffer for received CAN data from Legacy Rx FIFO.

flexcan_fd_frame_t *volatile rxFifoFDFrameBuf

The buffer for received CAN FD data from Enhanced Rx FIFO.

size_t rxFifoFrameNum

The number of CAN messages remaining to be received from Legacy or Enhanced Rx FIFO.

size_t rxFifoTransferTotalNum

Total CAN Message number need to be received from Legacy or Enhanced Rx FIFO.

volatile uint8_t mbState[CAN_WORD1_COUNT]

Message Buffer transfer state.

volatile uint8_t rxFifoState

Rx FIFO transfer state.

volatile uint32_t timestamp[CAN_WORD1_COUNT]

Mailbox transfer timestamp.

struct byteStatus

Public Members

bool byteIsRead

The byte n (0~3) was read or not. The type of error and which bit in byte (n) is affected by the error.

struct ____unnamed66____

Public Members

uint32_t timestamp

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t length

CAN frame data length in bytes (Range: 0~8).

uint32_t type

CAN Frame Type(DATA or REMOTE).

uint32_t format

CAN Frame Identifier(STD or EXT format).

uint32_t __pad0__

Reserved.

uint32_t idhit

CAN Rx FIFO filter hit id(This value is only used in Rx FIFO receive mode).

struct __unnamed68__

Public Members

uint32_t id

CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

uint32_t __pad0__

Reserved.

union __unnamed70__

Public Members

struct _flexcan_frame

struct _flexcan_frame

struct __unnamed72__

Public Members

uint32_t dataWord0

CAN Frame payload word0.

uint32_t dataWord1

CAN Frame payload word1.

struct __unnamed74__

Public Members

uint8_t dataByte3

CAN Frame payload byte3.

uint8_t dataByte2

CAN Frame payload byte2.

```
uint8_t dataByte1
    CAN Frame payload byte1.
uint8_t dataByte0
    CAN Frame payload byte0.
uint8_t dataByte7
    CAN Frame payload byte7.
uint8_t dataByte6
    CAN Frame payload byte6.
uint8_t dataByte5
    CAN Frame payload byte5.
uint8_t dataByte4
    CAN Frame payload byte4.
struct ____unnamed76____
```

Public Members

```
uint32_t timestamp
    FlexCAN internal Free-Running Counter Time Stamp.
uint32_t length
    CAN FD frame data length code (DLC), range see _flexcan_fd_frame_length, When the
    length <= 8, it equal to the data length, otherwise the number of valid frame data is
    not equal to the length value. user can use DLC_LENGTH_DECODE(length) macro to
    get the number of valid data bytes.
uint32_t type
    CAN Frame Type(DATA only).
uint32_t format
    CAN Frame Identifier(STD or EXT format).
uint32_t srr
    Substitute Remote request.
uint32_t esi
    Error State Indicator.
uint32_t brs
    Bit Rate Switch.
uint32_t edl
    Extended Data Length.
struct ____unnamed78____
```

Public Members

```
uint32_t id
    CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD()
    macro.
uint32_t ____pad0____
    Reserved.
union ____unnamed80____
```


Public Members

struct __flexcan_fd_frame

struct __flexcan_fd_frame

struct ____unnamed82____

Public Members

uint32_t dataWord[16]

CAN FD Frame payload, 16 double word maximum.

struct ____unnamed84____

Public Members

uint8_t dataByte3

CAN Frame payload byte3.

uint8_t dataByte2

CAN Frame payload byte2.

uint8_t dataByte1

CAN Frame payload byte1.

uint8_t dataByte0

CAN Frame payload byte0.

uint8_t dataByte7

CAN Frame payload byte7.

uint8_t dataByte6

CAN Frame payload byte6.

uint8_t dataByte5

CAN Frame payload byte5.

uint8_t dataByte4

CAN Frame payload byte4.

union ____unnamed86____

Public Members

struct __flexcan_config

struct __flexcan_config

struct ____unnamed88____

Public Members

uint32_t baudRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t baudRateFD

FlexCAN FD bit rate in bps, for CANFD data phase.

struct ____unnamed90____

Public Members

uint32_t bitRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t bitRateFD

FlexCAN FD bit rate in bps, for CANFD data phase.

union ____unnamed92____

Public Members

struct __flexcan_pn_config

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range detection”.

struct __flexcan_pn_config

struct ____unnamed96____

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range detection”.

Public Members

uint32_t lowerWord0

CAN Frame payload word0.

uint32_t lowerWord1

CAN Frame payload word1.

struct ____unnamed98____

Public Members

uint8_t lowerByte3

CAN Frame payload byte3.

uint8_t lowerByte2

CAN Frame payload byte2.

uint8_t lowerByte1

CAN Frame payload byte1.

uint8_t lowerByte0

CAN Frame payload byte0.

uint8_t lowerByte7

CAN Frame payload byte7.

uint8_t lowerByte6

CAN Frame payload byte6.

uint8_t lowerByte5

CAN Frame payload byte5.

```
uint8_t lowerByte4
    CAN Frame payload byte4.
union __unnamed94__
```

Public Members

```
struct __flexcan_pn_config
    < The data target values 2 which used only as the upper limit value in data match
    "range
    detection" or used to store the data mask in "equal to".
struct __flexcan_pn_config
struct __unnamed100__
    < The data target values 2 which used only as the upper limit value in data match "range
    detection" or used to store the data mask in "equal to".
```

Public Members

```
uint32_t upperWord0
    CAN Frame payload word0.
uint32_t upperWord1
    CAN Frame payload word1.
struct __unnamed102__
```

Public Members

```
uint8_t upperByte3
    CAN Frame payload byte3.
uint8_t upperByte2
    CAN Frame payload byte2.
uint8_t upperByte1
    CAN Frame payload byte1.
uint8_t upperByte0
    CAN Frame payload byte0.
uint8_t upperByte7
    CAN Frame payload byte7.
uint8_t upperByte6
    CAN Frame payload byte6.
uint8_t upperByte5
    CAN Frame payload byte5.
uint8_t upperByte4
    CAN Frame payload byte4.
```

2.18 FlexCAN eDMA Driver

```
void FLEXCAN_TransferCreateHandleEDMA(CAN_Type *base, flexcan_edma_handle_t *handle,  
                                       flexcan_edma_transfer_callback_t callback, void  
                                       *userData, edma_handle_t *rxFifoEdmaHandle)
```

Initializes the FlexCAN handle, which is used in transactional functions.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxFifoEdmaHandle – User-requested DMA handle for Rx FIFO DMA transfer.

```
void FLEXCAN_PrepareTransfConfiguration(CAN_Type *base, flexcan_fifo_transfer_t *pFifoXfer,  
                                       edma_transfer_config_t *pEdmaConfig)
```

Prepares the eDMA transfer configuration for FLEXCAN Legacy RX FIFO.

This function prepares the eDMA transfer configuration structure according to FLEXCAN Legacy RX FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

```
status_t FLEXCAN_StartTransferDatafromRxFIFO(CAN_Type *base, flexcan_edma_handle_t  
                                             *handle, edma_transfer_config_t  
                                             *pEdmaConfig)
```

Start Transfer Data from the FLEXCAN Legacy Rx FIFO using eDMA.

This function to Update edma transfer configuration and Start eDMA transfer

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

```
status_t FLEXCAN_TransferReceiveFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle,  
                                          flexcan_fifo_transfer_t *pFifoXfer)
```

Receives the CAN Message from the Legacy Rx FIFO using eDMA.

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.

- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flex-can_fifo_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

status_t FLEXCAN_TransferGetReceiveFifoCountEMDA(CAN_Type *base, flexcan_edma_handle_t *handle, size_t *count)

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

void FLEXCAN_TransferAbortReceiveFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle)

Aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

This function aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.

status_t FLEXCAN_TransferReceiveEnhancedFifoEDMA(CAN_Type *base, flexcan_edma_handle_t *handle, flexcan_fifo_transfer_t *pFifoXfer)

Receives the CAN FD Message from the Enhanced Rx FIFO using eDMA.

This function receives the CAN FD Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flex-can_fifo_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

static inline *status_t* FLEXCAN_TransferGetReceiveEnhancedFifoCountEMDA(CAN_Type *base, flexcan_edma_handle_t *handle, size_t *count)

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- `base` – FlexCAN peripheral base address.
- `handle` – FlexCAN handle pointer.
- `count` – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

FSL_FLEXCAN_EDMA_DRIVER_VERSION

FlexCAN EDMA driver version.

```
typedef struct flexcan_edma_handle flexcan_edma_handle_t
```

```
typedef void (*flexcan_edma_transfer_callback_t)(CAN_Type *base, flexcan_edma_handle_t *handle, status_t status, void *userData)
```

FlexCAN transfer callback function.

```
struct flexcan_edma_handle
```

```
#include <fsl_flexcan_edma.h> FlexCAN eDMA handle.
```

Public Members

flexcan_edma_transfer_callback_t callback

Callback function.

void *userData

FlexCAN callback function parameter.

edma_handle_t *rxFifoEdmaHandle

The EDMA handler for Rx FIFO.

volatile uint8_t rxFifoState

Rx FIFO transfer state.

size_t frameNum

The number of messages that need to be received.

flexcan_fd_frame_t *framefd

Point to the buffer of CAN Message to be received from Enhanced Rx FIFO.

2.19 FlexIO: FlexIO Driver

2.20 FlexIO Driver

```
void FLEXIO_GetDefaultConfig(flexio_config_t *userConfig)
```

Gets the default configuration to configure the FlexIO module. The configuration can used directly to call the FLEXIO_Configure().

Example:

```
flexio_config_t config;  
FLEXIO_GetDefaultConfig(&config);
```

Parameters

- userConfig – pointer to flexio_config_t structure

void FLEXIO_Init(FLEXIO_Type *base, const flexio_config_t *userConfig)

Configures the FlexIO with a FlexIO configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_GetDefaultConfig().

Example

```
flexio_config_t config = {  
    .enableFlexio = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false  
};  
FLEXIO_Configure(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- userConfig – pointer to flexio_config_t structure

void FLEXIO_Deinit(FLEXIO_Type *base)

Gates the FlexIO clock. Call this API to stop the FlexIO clock.

Note: After calling this API, call the FLEXIO_Init to use the FlexIO module.

Parameters

- base – FlexIO peripheral base address

uint32_t FLEXIO_GetInstance(FLEXIO_Type *base)

Get instance number for FLEXIO module.

Parameters

- base – FLEXIO peripheral base address.

void FLEXIO_Reset(FLEXIO_Type *base)

Resets the FlexIO module.

Parameters

- base – FlexIO peripheral base address

static inline void FLEXIO_Enable(FLEXIO_Type *base, bool enable)

Enables the FlexIO module operation.

Parameters

- base – FlexIO peripheral base address
- enable – true to enable, false to disable.

static inline uint32_t FLEXIO_ReadPinInput(FLEXIO_Type *base)

Reads the input data on each of the FlexIO pins.

Parameters

- base – FlexIO peripheral base address

Returns

FlexIO pin input data

```
static inline uint8_t FLEXIO_GetShifterState(FLEXIO_Type *base)
```

Gets the current state pointer for state mode use.

Parameters

- base – FlexIO peripheral base address

Returns

current State pointer

```
void FLEXIO_SetShifterConfig(FLEXIO_Type *base, uint8_t index, const flexio_shifter_config_t *shifterConfig)
```

Configures the shifter with the shifter configuration. The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {  
.timerSelect = 0,  
.timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,  
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,  
.pinPolarity = kFLEXIO_PinActiveLow,  
.shifterMode = kFLEXIO_ShifterModeTransmit,  
.inputSource = kFLEXIO_ShifterInputFromPin,  
.shifterStop = kFLEXIO_ShifterStopBitHigh,  
.shifterStart = kFLEXIO_ShifterStartBitLow  
};  
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Shifter index
- shifterConfig – Pointer to flexio_shifter_config_t structure

```
void FLEXIO_SetTimerConfig(FLEXIO_Type *base, uint8_t index, const flexio_timer_config_t *timerConfig)
```

Configures the timer with the timer configuration. The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

Example

```
flexio_timer_config_t config = {  
.triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFToNSTAT(0),  
.triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,  
.triggerSource = kFLEXIO_TimerTriggerSourceInternal,  
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,  
.pinSelect = 0,  
.pinPolarity = kFLEXIO_PinActiveHigh,  
.timerMode = kFLEXIO_TimerModeDual8BitBaudBit,  
.timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,  
.timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput,  
.timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,  
.timerDisable = kFLEXIO_TimerDisableOnTimerCompare,  
.timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,  
.timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,  
.timerStart = kFLEXIO_TimerStartBitEnabled  
};  
FLEXIO_SetTimerConfig(base, &config);
```


Parameters

- base – FlexIO peripheral base address
- index – Timer index
- timerConfig – Pointer to the flexio_timer_config_t structure

```
static inline void FLEXIO_SetClockMode(FLEXIO_Type *base, uint8_t index,  
                                       flexio_timer_decrement_source_t clocksource)
```

This function set the value of the prescaler on flexio channels.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- index – Timer index
- clocksource – Set clock value

```
static inline void FLEXIO_EnableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

```
static inline void FLEXIO_DisableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Disables the shifter status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

```
static inline void FLEXIO_EnableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
```

Enables the shifter error interrupt. The interrupt generates when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_DisableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the shifter error interrupt. The interrupt won't generate when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_EnableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Enables the timer status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_DisableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the timer status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline uint32_t FLEXIO_GetShifterStatusFlags(FLEXIO_Type *base)
Gets the shifter status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter status flags

static inline void FLEXIO_ClearShifterStatusFlags(FLEXIO_Type *base, uint32_t mask)
Clears the shifter status flags.

Note: For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address

- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetShifterErrorFlags(FLEXIO_Type *base)

Gets the shifter error flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter error flags

static inline void FLEXIO_ClearShifterErrorFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter error flags.

Note: For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline uint32_t FLEXIO_GetTimerStatusFlags(FLEXIO_Type *base)

Gets the timer status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Timer status flags

static inline void FLEXIO_ClearTimerStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the timer status flags.

Note: For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_EnableShifterStatusDMA(FLEXIO_Type *base, uint32_t mask, bool enable)

Enables/disables the shifter status DMA. The DMA request generates when the corresponding SSF is set.

Note: For multiple shifter status DMA enables, for example, calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

- enable – True to enable, false to disable.

uint32_t FLEXIO_GetShifterBufferAddress(FLEXIO_Type *base, flexio_shifter_buffer_type_t type, uint8_t index)

Gets the shifter buffer address for the DMA transfer usage.

Parameters

- base – FlexIO peripheral base address
- type – Shifter type of flexio_shifter_buffer_type_t
- index – Shifter index

Returns

Corresponding shifter buffer index

status_t FLEXIO_RegisterHandleIRQ(void *base, void *handle, flexio_isr_t isr)

Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- handle – Pointer to the handler for FlexIO simulated peripheral.
- isr – FlexIO simulated peripheral interrupt handler.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_UnregisterHandleIRQ(void *base)

Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

static inline void FLEXIO_ClearPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 0.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_SetPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 1.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_TogglePortOutput(FLEXIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple FLEXIO pins.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_PinWrite(FLEXIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the FLEXIO pins to the logic 1 or 0.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- output – FLEXIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void FLEXIO_EnablePinOutput(FLEXIO_Type *base, uint32_t pin)

Enables the FLEXIO output pin function.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

static inline uint32_t FLEXIO_PinRead(FLEXIO_Type *base, uint32_t pin)

Reads the current input value of the FLEXIO pin.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline uint32_t FLEXIO_GetPinStatus(FLEXIO_Type *base, uint32_t pin)

Gets the FLEXIO input pin status.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input status

- 0: corresponding pin input capture no status.
- 1: corresponding pin input capture rising or falling edge.

static inline void FLEXIO_SetPinLevel(FLEXIO_Type *base, uint8_t pin, bool level)

Sets the FLEXIO output pin level.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.
- level – FlexIO output pin level to set, can be either 0 or 1.

static inline bool FLEXIO_GetPinOverride(const FLEXIO_Type *const base, uint8_t pin)

Gets the enabled status of a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address

- pin – FlexIO pin number.

Return values

FlexIO – port enabled status

- 0: corresponding output pin is in disabled state.
- 1: corresponding output pin is in enabled state.

static inline void FLEXIO_ConfigPinOverride(FLEXIO_Type *base, uint8_t pin, bool enabled)
Enables or disables a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – Flexio pin number.
- enabled – Enable or disable the FlexIO pin.

static inline void FLEXIO_ClearPortStatus(FLEXIO_Type *base, uint32_t mask)
Clears the multiple FLEXIO input pins status.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

FSL_FLEXIO_DRIVER_VERSION
FlexIO driver version.

enum _flexio_timer_trigger_polarity
Define time of timer trigger polarity.

Values:

enumerator kFLEXIO_TimerTriggerPolarityActiveHigh
Active high.

enumerator kFLEXIO_TimerTriggerPolarityActiveLow
Active low.

enum _flexio_timer_trigger_source
Define type of timer trigger source.

Values:

enumerator kFLEXIO_TimerTriggerSourceExternal
External trigger selected.

enumerator kFLEXIO_TimerTriggerSourceInternal
Internal trigger selected.

enum _flexio_pin_config
Define type of timer/shifter pin configuration.

Values:

enumerator kFLEXIO_PinConfigOutputDisabled
Pin output disabled.

enumerator kFLEXIO_PinConfigOpenDrainOrBidirection
Pin open drain or bidirectional output enable.

enumerator kFLEXIO_PinConfigBidirectionOutputData
Pin bidirectional output data.

enumerator kFLEXIO_PinConfigOutput

Pin output.

enum _flexio_pin_polarity

Definition of pin polarity.

Values:

enumerator kFLEXIO_PinActiveHigh

Active high.

enumerator kFLEXIO_PinActiveLow

Active low.

enum _flexio_timer_mode

Define type of timer work mode.

Values:

enumerator kFLEXIO_TimerModeDisabled

Timer Disabled.

enumerator kFLEXIO_TimerModeDual8BitBaudBit

Dual 8-bit counters baud/bit mode.

enumerator kFLEXIO_TimerModeDual8BitPWM

Dual 8-bit counters PWM mode.

enumerator kFLEXIO_TimerModeSingle16Bit

Single 16-bit counter mode.

enumerator kFLEXIO_TimerModeDual8BitPWMLow

Dual 8-bit counters PWM Low mode.

enum _flexio_timer_output

Define type of timer initial output or timer reset condition.

Values:

enumerator kFLEXIO_TimerOutputOneNotAffectedByReset

Logic one when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputZeroNotAffectedByReset

Logic zero when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputOneAffectedByReset

Logic one when enabled and on timer reset.

enumerator kFLEXIO_TimerOutputZeroAffectedByReset

Logic zero when enabled and on timer reset.

enum _flexio_timer_decrement_source

Define type of timer decrement.

Values:

enumerator kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput

Decrement counter on FlexIO clock, Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput

Decrement counter on Trigger input (both edges), Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnPinInputShiftPinInput

Decrement counter on Pin input (both edges), Shift clock equals Pin input.

enumerator kFLEXIO__TimerDecSrcOnTriggerInputShiftTriggerInput

Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

enum __flexio_timer_reset_condition

Define type of timer reset condition.

Values:

enumerator kFLEXIO__TimerResetNever

Timer never reset.

enumerator kFLEXIO__TimerResetOnTimerPinEqualToTimerOutput

Timer reset on Timer Pin equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerTriggerEqualToTimerOutput

Timer reset on Timer Trigger equal to Timer Output.

enumerator kFLEXIO__TimerResetOnTimerPinRisingEdge

Timer reset on Timer Pin rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerRisingEdge

Timer reset on Trigger rising edge.

enumerator kFLEXIO__TimerResetOnTimerTriggerBothEdge

Timer reset on Trigger rising or falling edge.

enum __flexio_timer_disable_condition

Define type of timer disable condition.

Values:

enumerator kFLEXIO__TimerDisableNever

Timer never disabled.

enumerator kFLEXIO__TimerDisableOnPreTimerDisable

Timer disabled on Timer N-1 disable.

enumerator kFLEXIO__TimerDisableOnTimerCompare

Timer disabled on Timer compare.

enumerator kFLEXIO__TimerDisableOnTimerCompareTriggerLow

Timer disabled on Timer compare and Trigger Low.

enumerator kFLEXIO__TimerDisableOnPinBothEdge

Timer disabled on Pin rising or falling edge.

enumerator kFLEXIO__TimerDisableOnPinBothEdgeTriggerHigh

Timer disabled on Pin rising or falling edge provided Trigger is high.

enumerator kFLEXIO__TimerDisableOnTriggerFallingEdge

Timer disabled on Trigger falling edge.

enum __flexio_timer_enable_condition

Define type of timer enable condition.

Values:

enumerator kFLEXIO__TimerEnabledAlways

Timer always enabled.

enumerator kFLEXIO__TimerEnableOnPrevTimerEnable

Timer enabled on Timer N-1 enable.

enumerator kFLEXIO__TimerEnableOnTriggerHigh

Timer enabled on Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerHighPinHigh

Timer enabled on Trigger high and Pin high.

enumerator kFLEXIO__TimerEnableOnPinRisingEdge

Timer enabled on Pin rising edge.

enumerator kFLEXIO__TimerEnableOnPinRisingEdgeTriggerHigh

Timer enabled on Pin rising edge and Trigger high.

enumerator kFLEXIO__TimerEnableOnTriggerRisingEdge

Timer enabled on Trigger rising edge.

enumerator kFLEXIO__TimerEnableOnTriggerBothEdge

Timer enabled on Trigger rising or falling edge.

enum _flexio_timer_stop_bit_condition

Define type of timer stop bit generate condition.

Values:

enumerator kFLEXIO__TimerStopBitDisabled

Stop bit disabled.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompare

Stop bit is enabled on timer compare.

enumerator kFLEXIO__TimerStopBitEnableOnTimerDisable

Stop bit is enabled on timer disable.

enumerator kFLEXIO__TimerStopBitEnableOnTimerCompareDisable

Stop bit is enabled on timer compare and timer disable.

enum _flexio_timer_start_bit_condition

Define type of timer start bit generate condition.

Values:

enumerator kFLEXIO__TimerStartBitDisabled

Start bit disabled.

enumerator kFLEXIO__TimerStartBitEnabled

Start bit enabled.

enum _flexio_timer_output_state

FlexIO as PWM channel output state.

Values:

enumerator kFLEXIO__PwmLow

The output state of PWM channel is low

enumerator kFLEXIO__PwmHigh

The output state of PWM channel is high

enum _flexio_shifter_timer_polarity

Define type of timer polarity for shifter control.

Values:

enumerator kFLEXIO__ShifterTimerPolarityOnPositive

Shift on positive edge of shift clock.

enumerator kFLEXIO__ShifterTimerPolarityOnNegative
Shift on negative edge of shift clock.

enum _flexio__shifter__mode

Define type of shifter working mode.

Values:

enumerator kFLEXIO__ShifterDisabled
Shifter is disabled.

enumerator kFLEXIO__ShifterModeReceive
Receive mode.

enumerator kFLEXIO__ShifterModeTransmit
Transmit mode.

enumerator kFLEXIO__ShifterModeMatchStore
Match store mode.

enumerator kFLEXIO__ShifterModeMatchContinuous
Match continuous mode.

enumerator kFLEXIO__ShifterModeState
SHIFTBUF contents are used for storing programmable state attributes.

enumerator kFLEXIO__ShifterModeLogic
SHIFTBUF contents are used for implementing programmable logic look up table.

enum _flexio__shifter__input__source

Define type of shifter input source.

Values:

enumerator kFLEXIO__ShifterInputFromPin
Shifter input from pin.

enumerator kFLEXIO__ShifterInputFromNextShifterOutput
Shifter input from Shifter N+1.

enum _flexio__shifter__stop__bit

Define of STOP bit configuration.

Values:

enumerator kFLEXIO__ShifterStopBitDisable
Disable shifter stop bit.

enumerator kFLEXIO__ShifterStopBitLow
Set shifter stop bit to logic low level.

enumerator kFLEXIO__ShifterStopBitHigh
Set shifter stop bit to logic high level.

enum _flexio__shifter__start__bit

Define type of START bit configuration.

Values:

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnEnable
Disable shifter start bit, transmitter loads data on enable.

enumerator kFLEXIO__ShifterStartBitDisabledLoadDataOnShift
Disable shifter start bit, transmitter loads data on first shift.

enumerator kFLEXIO__ShifterStartBitLow

Set shifter start bit to logic low level.

enumerator kFLEXIO__ShifterStartBitHigh

Set shifter start bit to logic high level.

enum __flexio__shifter__buffer__type

Define FlexIO shifter buffer type.

Values:

enumerator kFLEXIO__ShifterBuffer

Shifter Buffer N Register.

enumerator kFLEXIO__ShifterBufferBitSwapped

Shifter Buffer N Bit Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferByteSwapped

Shifter Buffer N Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferBitByteSwapped

Shifter Buffer N Bit Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleByteSwapped

Shifter Buffer N Nibble Byte Swapped Register.

enumerator kFLEXIO__ShifterBufferHalfWordSwapped

Shifter Buffer N Half Word Swapped Register.

enumerator kFLEXIO__ShifterBufferNibbleSwapped

Shifter Buffer N Nibble Swapped Register.

enum __flexio__gpio__direction

FLEXIO gpio direction definition.

Values:

enumerator kFLEXIO__DigitalInput

Set current pin as digital input

enumerator kFLEXIO__DigitalOutput

Set current pin as digital output

enum __flexio__pin__input__config

FLEXIO gpio input config.

Values:

enumerator kFLEXIO__InputInterruptDisabled

Interrupt request is disabled.

enumerator kFLEXIO__InputInterruptEnable

Interrupt request is enable.

enumerator kFLEXIO__FlagRisingEdgeEnable

Input pin flag on rising edge.

enumerator kFLEXIO__FlagFallingEdgeEnable

Input pin flag on falling edge.

typedef enum __flexio_timer_trigger_polarity flexio__timer__trigger__polarity_t

Define time of timer trigger polarity.

`typedef enum _flexio_timer_trigger_source flexio_timer_trigger_source_t`
Define type of timer trigger source.

`typedef enum _flexio_pin_config flexio_pin_config_t`
Define type of timer/shifter pin configuration.

`typedef enum _flexio_pin_polarity flexio_pin_polarity_t`
Definition of pin polarity.

`typedef enum _flexio_timer_mode flexio_timer_mode_t`
Define type of timer work mode.

`typedef enum _flexio_timer_output flexio_timer_output_t`
Define type of timer initial output or timer reset condition.

`typedef enum _flexio_timer_decrement_source flexio_timer_decrement_source_t`
Define type of timer decrement.

`typedef enum _flexio_timer_reset_condition flexio_timer_reset_condition_t`
Define type of timer reset condition.

`typedef enum _flexio_timer_disable_condition flexio_timer_disable_condition_t`
Define type of timer disable condition.

`typedef enum _flexio_timer_enable_condition flexio_timer_enable_condition_t`
Define type of timer enable condition.

`typedef enum _flexio_timer_stop_bit_condition flexio_timer_stop_bit_condition_t`
Define type of timer stop bit generate condition.

`typedef enum _flexio_timer_start_bit_condition flexio_timer_start_bit_condition_t`
Define type of timer start bit generate condition.

`typedef enum _flexio_timer_output_state flexio_timer_output_state_t`
FlexIO as PWM channel output state.

`typedef enum _flexio_shifter_timer_polarity flexio_shifter_timer_polarity_t`
Define type of timer polarity for shifter control.

`typedef enum _flexio_shifter_mode flexio_shifter_mode_t`
Define type of shifter working mode.

`typedef enum _flexio_shifter_input_source flexio_shifter_input_source_t`
Define type of shifter input source.

`typedef enum _flexio_shifter_stop_bit flexio_shifter_stop_bit_t`
Define of STOP bit configuration.

`typedef enum _flexio_shifter_start_bit flexio_shifter_start_bit_t`
Define type of START bit configuration.

`typedef enum _flexio_shifter_buffer_type flexio_shifter_buffer_type_t`
Define FlexIO shifter buffer type.

`typedef struct _flexio_config flexio_config_t`
Define FlexIO user configuration structure.

`typedef struct _flexio_timer_config flexio_timer_config_t`
Define FlexIO timer configuration structure.

`typedef struct _flexio_shifter_config flexio_shifter_config_t`
Define FlexIO shifter configuration structure.

typedef enum *flexio_gpio_direction* flexio_gpio_direction_t
FLEXIO gpio direction definition.

typedef enum *flexio_pin_input_config* flexio_pin_input_config_t
FLEXIO gpio input config.

typedef struct *flexio_gpio_config* flexio_gpio_config_t
The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

typedef void (*flexio_isr_t)(void *base, void *handle)
typedef for FlexIO simulated driver interrupt handler.

FLEXIO_Type *const s_flexioBases[]
Pointers to flexio bases for each instance.

const clock_ip_name_t s_flexioClocks[]
Pointers to flexio clocks for each instance.

void FLEXIO_SetPinConfig(FLEXIO_Type *base, uint32_t pin, *flexio_gpio_config_t* *config)
Configure a FLEXIO pin used by the board.

To Config the FLEXIO PIN, define a pin configuration, as either input or output, in the user file. Then, call the FLEXIO_SetPinConfig() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalInput,
    0U,
    kFLEXIO_FlagRisingEdgeEnable | kFLEXIO_InputInterruptEnable,
}
Define a digital output pin configuration,
flexio_gpio_config_t config =
{
    kFLEXIO_DigitalOutput,
    0U,
    0U
}
```

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- config – FLEXIO pin configuration pointer.

FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x)

Calculate FlexIO timer trigger.

FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(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

2.21 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

2.22 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

2.23 FlexIO I2C Master Driver

status_t FLEXIO_I2C_CheckForBusyBus(*FLEXIO_I2C_Type* *base)

Make sure the bus isn't already pulled down.

Check the FLEXIO pin status to see whether either of SDA and SCL pin is pulled down.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure..

Return values

- kStatus_Success –
- kStatus_FLEXIO_I2C_Busy –

status_t FLEXIO_I2C_MasterInit(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_config_t* *masterConfig, *uint32_t* srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.

Example

```
FLEXIO_I2C_Type base = {  
.flexioBase = FLEXIO,  
.SDAPinIndex = 0,  
.SCLPinIndex = 1,  
.shifterIndex = {0,1},  
.timerIndex = {0,1}  
};  
flexio_i2c_master_config_t config = {  
.enableInDoze = false,  
.enableInDebug = true,  
.enableFastAccess = false,  
.baudRate_Bps = 100000  
};  
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- masterConfig – Pointer to flexio_i2c_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Initialization successful
- kStatus_InvalidArgument – The source clock exceed upper range limitation

void FLEXIO_I2C_MasterDeinit(*FLEXIO_I2C_Type* *base)

De-initializes the FlexIO I2C master peripheral. Calling this API Resets the FlexIO I2C master shifer and timer config, module can't work unless the FLEXIO_I2C_MasterInit is called.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterGetDefaultConfig(*flexio_i2c_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO module. The configuration can be used directly for calling the FLEXIO_I2C_MasterInit().

Example:

```
flexio_i2c_master_config_t config;
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – Pointer to flexio_i2c_master_config_t structure.

static inline void FLEXIO_I2C_MasterEnable(*FLEXIO_I2C_Type* *base, bool enable)
Enables/disables the FlexIO module operation.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – Pass true to enable module, false does not have any effect.

uint32_t FLEXIO_I2C_MasterGetStatusFlags(*FLEXIO_I2C_Type* *base)
Gets the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure

Returns

Status flag, use status flag to AND _flexio_i2c_master_status_flags can get the related status.

void FLEXIO_I2C_MasterClearStatusFlags(*FLEXIO_I2C_Type* *base, uint32_t mask)
Clears the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
 - kFLEXIO_I2C_RxFullFlag
 - kFLEXIO_I2C_ReceiveNakFlag

void FLEXIO_I2C_MasterEnableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)
Enables the FlexIO i2c master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source. Currently only one interrupt request source:
 - kFLEXIO_I2C_TransferCompleteInterruptEnable

void FLEXIO_I2C_MasterDisableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)
Disables the FlexIO I2C master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source.

void FLEXIO_I2C_MasterSetBaudRate(*FLEXIO_I2C_Type* *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the FlexIO I2C master transfer baudrate.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- baudRate_Bps – the baud rate value in HZ

- srcClock_Hz – source clock in HZ

void FLEXIO_I2C_MasterStart(*FLEXIO_I2C_Type* *base, uint8_t address, *flexio_i2c_direction_t* direction)

Sends START + 7-bit address to the bus.

Note: This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO_I2C_RxFullFlag status is asserted before calling this API.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- address – 7-bit address.
- direction – transfer direction. This parameter is one of the values in flexio_i2c_direction_t:
 - kFLEXIO_I2C_Write: Transmit
 - kFLEXIO_I2C_Read: Receive

void FLEXIO_I2C_MasterStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterRepeatedStart(*FLEXIO_I2C_Type* *base)

Sends the repeated start signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterAbortStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal when transfer is still on-going.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterEnableAck(*FLEXIO_I2C_Type* *base, bool enable)

Configures the sent ACK/NAK for the following byte.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – True to configure send ACK, false configure to send NAK.

status_t FLEXIO_I2C_MasterSetTransferCount(*FLEXIO_I2C_Type* *base, uint16_t count)

Sets the number of bytes to be transferred from a start signal to a stop signal.

Note: Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

- `count` – Number of bytes need to be transferred from a start signal to a re-start/stop signal

Return values

- `kStatus_Success` – Successfully configured the count.
- `kStatus_InvalidArgument` – Input argument is invalid.

```
static inline void FLEXIO_I2C_MasterWriteByte(FLEXIO_I2C_Type *base, uint32_t data)
```

Writes one byte of data to the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the `TxEmptyFlag` is asserted before calling this API.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `data` – a byte of data.

```
static inline uint8_t FLEXIO_I2C_MasterReadByte(FLEXIO_I2C_Type *base)
```

Reads one byte of data from the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.

Returns

data byte read.

```
status_t FLEXIO_I2C_MasterWriteBlocking(FLEXIO_I2C_Type *base, const uint8_t *txBuff,  
                                         uint8_t txSize)
```

Sends a buffer of data in bytes.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- `base` – Pointer to `FLEXIO_I2C_Type` structure.
- `txBuff` – The data bytes to send.
- `txSize` – The number of data bytes to send.

Return values

- `kStatus_Success` – Successfully write data.
- `kStatus_FLEXIO_I2C_Nak` – Receive NAK during writing data.
- `kStatus_FLEXIO_I2C_Timeout` – Timeout polling status flags.

```
status_t FLEXIO_I2C_MasterReadBlocking(FLEXIO_I2C_Type *base, uint8_t *rxBuff, uint8_t  
                                         rxSize)
```

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- rxBuff – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus__Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

status_t FLEXIO_I2C_MasterTransferBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.
- xfer – pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

status_t FLEXIO_I2C_MasterTransferCreateHandle(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_callback_t
callback, void *userData)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure to store the transfer state.
- callback – Pointer to user callback function.
- userData – User param passed to the callback function.

Return values

- kStatus__Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/isr table out of range.

status_t FLEXIO_I2C_MasterTransferNonBlocking(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle,
flexio_i2c_master_transfer_t *xfer)

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: The API returns immediately after the transfer initiates. Call FLEXIO_I2C_MasterTransferGetCount to poll the transfer status to check whether the

transfer is finished. If the return status is not kStatus_FLEXIO_I2C_Busy, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state
- xfer – pointer to flexio_i2c_master_transfer_t structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_FLEXIO_I2C_Busy – FlexIO I2C is not idle, is running another transfer.

status_t FLEXIO_I2C_MasterTransferGetCount(*FLEXIO_I2C_Type* *base,
flexio_i2c_master_handle_t *handle, size_t
*count)

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.
- kStatus_Success – Successfully return the count.

void FLEXIO_I2C_MasterTransferAbort(*FLEXIO_I2C_Type* *base, flexio_i2c_master_handle_t
*handle)

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state

void FLEXIO_I2C_MasterTransferHandleIRQ(void *i2cType, void *i2cHandle)

Master interrupt handler.

Parameters

- i2cType – Pointer to FLEXIO_I2C_Type structure
- i2cHandle – Pointer to flexio_i2c_master_transfer_t structure

FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION

FlexIO I2C transfer status.

Values:

enumerator kStatus_FLEXIO_I2C_Busy

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Idle

I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Nak

NAK received during transfer.

enumerator kStatus_FLEXIO_I2C_Timeout

Timeout polling status flags.

enum _flexio_i2c_master_interrupt

Define FlexIO I2C master interrupt mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyInterruptEnable

Tx buffer empty interrupt enable.

enumerator kFLEXIO_I2C_RxFullInterruptEnable

Rx buffer full interrupt enable.

enum _flexio_i2c_master_status_flags

Define FlexIO I2C master status mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyFlag

Tx shifter empty flag.

enumerator kFLEXIO_I2C_RxFullFlag

Rx shifter full/Transfer complete flag.

enumerator kFLEXIO_I2C_ReceiveNakFlag

Receive NAK flag.

enum _flexio_i2c_direction

Direction of master transfer.

Values:

enumerator kFLEXIO_I2C_Write

Master send to slave.

enumerator kFLEXIO_I2C_Read

Master receive from slave.

typedef enum _flexio_i2c_direction flexio_i2c_direction_t

Direction of master transfer.

typedef struct _flexio_i2c_type FLEXIO_I2C_Type

Define FlexIO I2C master access structure typedef.

typedef struct _flexio_i2c_master_config flexio_i2c_master_config_t

Define FlexIO I2C master user configuration structure.

```
typedef struct _flexio_i2c_master_transfer flexio_i2c_master_transfer_t
```

Define FlexIO I2C master transfer structure.

```
typedef struct _flexio_i2c_master_handle flexio_i2c_master_handle_t
```

FlexIO I2C master handle typedef.

```
typedef void (*flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type *base,  
flexio_i2c_master_handle_t *handle, status_t status, void *userData)
```

FlexIO I2C master transfer callback typedef.

```
I2C_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _flexio_i2c_type
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t SDAPinIndex

Pin select for I2C SDA.

uint8_t SCLPinIndex

Pin select for I2C SCL.

uint8_t shifterIndex[2]

Shifter index used in FlexIO I2C.

uint8_t timerIndex[3]

Timer index used in FlexIO I2C.

uint32_t baudrate

Master transfer baudrate, used to calculate delay time.

```
struct _flexio_i2c_master_config
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master user configuration structure.

Public Members

bool enableMaster

Enables the FlexIO I2C peripheral at initialization time.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

```
struct _flexio_i2c_master_transfer
```

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master transfer structure.

Public Members

uint32_t flags

Transfer flag which controls the transfer, reserved for FlexIO I2C.

uint8_t slaveAddress

7-bit slave address.

flexio_i2c_direction_t direction

Transfer direction, read or write.

uint32_t subaddress

Sub address. Transferred MSB first.

uint8_t subaddressSize

Size of sub address.

uint8_t volatile *data

Transfer buffer.

volatile size_t dataSize

Transfer size.

struct *_flexio_i2c_master_handle*

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master handle structure.

Public Members

flexio_i2c_master_transfer_t transfer

FlexIO I2C master transfer copy.

size_t transferSize

Total bytes to be transferred.

uint8_t state

Transfer state maintained during transfer.

flexio_i2c_master_transfer_callback_t completionCallback

Callback function called at transfer event. Callback function called at transfer event.

void *userData

Callback parameter passed to callback function.

bool needRestart

Whether master needs to send re-start signal.

2.24 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

2.25 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 register value backup.

2.26 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 ____unnamed168____

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.

2.27 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 void FLEXSPI_GetDataLearningPhase(FLEXSPI_Type *base, uint8_t *portAPhase, uint8_t *portBPhase)
```

Gets the sampling clock phase selection after Data Learning.

Parameters

- base – FLEXSPI peripheral base address.
- portAPhase – Pointer to a uint8_t type variable to receive the selected clock phase on PORTA.
- portBPhase – Pointer to a uint8_t type variable to receive the selected clock phase on PORTB.

```
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

```
static inline void FLEXSPI_EnableIPParallelMode(FLEXSPI_Type *base, bool enable)
```

Enables/disables the FLEXSPI IP command parallel mode.

Parameters

- base – FLEXSPI peripheral base address.
- enable – True means enable parallel mode, false means disable parallel mode.

```
static inline void FLEXSPI_EnableAHBParallelMode(FLEXSPI_Type *base, bool enable)
```

Enables/disables the FLEXSPI AHB command parallel mode.

Parameters

- base – FLEXSPI peripheral base address.
- enable – True means enable parallel mode, false means disable parallel mode.

```
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_AhbBusTimeoutFlag
AHB Bus timeout.

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_DataLearningFailedFlag
Data learning failed.

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.

`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.

2.28 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 kFLEXPSI_EDMASize1Bytes(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.

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

2.29 I3C: I3C Driver

FSL_I3C_DRIVER_VERSION

I3C driver version.

I3C status return codes.

Values:

enumerator kStatus_I3C_Busy

The master is already performing a transfer.

enumerator kStatus_I3C_Idle

The slave driver is idle.

enumerator kStatus_I3C_Nak

The slave device sent a NAK in response to an address.

enumerator kStatus_I3C_WriteAbort

The slave device sent a NAK in response to a write.

enumerator kStatus_I3C_Term

The master terminates slave read.

enumerator kStatus_I3C_HdrParityError

Parity error from DDR read.

enumerator kStatus_I3C_CrcError

CRC error from DDR read.

enumerator kStatus_I3C_ReadFifoError

Read from M/SRDATA register when FIFO empty.

enumerator kStatus_I3C_WriteFifoError

Write to M/SWDATA register when FIFO full.

enumerator kStatus_I3C_MsgError

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kStatus_I3C_InvalidReq

Invalid use of request.

enumerator kStatus_I3C_Timeout

The module has stalled too long in a frame.

enumerator kStatus_I3C_SlaveCountExceed

The I3C slave count has exceed the definition in I3C_MAX_DEVCNT.

enumerator kStatus_I3C_IBIWon

The I3C slave event IBI or MR or HJ won the arbitration on a header address.

enumerator kStatus_I3C_OverrunError

Slave internal from-bus buffer/FIFO overrun.

enumerator kStatus_I3C_UnderrunError

Slave internal to-bus buffer/FIFO underrun

enumerator kStatus_I3C_UnderrunNak

Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kStatus_I3C_InvalidStart

Slave invalid start flag

enumerator kStatus_I3C_SdrParityError

SDR parity error

enumerator kStatus_I3C_S0S1Error

S0 or S1 error

enum _i3c_hdr_mode

I3C HDR modes.

Values:

enumerator kI3C_HDRModeNone

enumerator kI3C_HDRModeDDR

enumerator kI3C_HDRModeTSP

enumerator kI3C_HDRModeTSL

typedef enum _i3c_hdr_mode i3c_hdr_mode_t

I3C HDR modes.

typedef struct _i3c_device_info i3c_device_info_t

I3C device information.

I3C_RETRY_TIMES

Max loops to wait for I3C operation status complete.

This is the maximum number of loops to wait for I3C operation status complete. If set to 0, it will wait indefinitely.

I3C_MAX_DEVCNT

I3C_IBI_BUFF_SIZE

struct _i3c_device_info

#include <fsl_i3c.h> I3C device information.

Public Members

uint8_t dynamicAddr

Device dynamic address.

uint8_t staticAddr

Static address.

uint8_t dcr

Device characteristics register information.

uint8_t bcr

Bus characteristics register information.

uint16_t vendorID

Device vendor ID(manufacture ID).

uint32_t partNumber
Device part number info

uint16_t maxReadLength
Maximum read length.

uint16_t maxWriteLength
Maximum write length.

uint8_t hdrMode
Support hdr mode, could be OR logic in i3c_hdr_mode.

2.30 I3C Common Driver

typedef struct *i3c_config* i3c_config_t

Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

uint32_t I3C_GetInstance(I3C_Type *base)

Get which instance current I3C is used.

Parameters

- base – The I3C peripheral base address.

void I3C_GetDefaultConfig(*i3c_config_t* *config)

Provides a default configuration for the I3C peripheral, the configuration covers both master functionality and slave functionality.

This function provides the following default configuration for I3C:

```
config->enableMaster      = kI3C_MasterCapable;
config->disableTimeout    = false;
config->hKeep              = kI3C_MasterHighKeeperNone;
config->enableOpenDrainStop = true;
config->enableOpenDrainHigh = true;
config->baudRate_Hz.i2cBaud = 400000U;
config->baudRate_Hz.i3cPushPullBaud = 12500000U;
config->baudRate_Hz.i3cOpenDrainBaud = 2500000U;
config->masterDynamicAddress = 0x0AU;
config->slowClock_Hz       = 1000000U;
config->enableSlave        = true;
config->vendorID            = 0x11BU;
config->enableRandomPart   = false;
config->partNumber          = 0;
config->dcr                 = 0;
config->bcr = 0;
config->hdrMode             = (uint8_t)kI3C_HDRModeDDR;
config->nakAllRequest       = false;
config->ignoreS0S1Error    = false;
config->offline             = false;
config->matchSlaveStartStop = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the common I3C driver with I3C_Init().

Parameters

- `config` – **[out]** User provided configuration structure for default values. Refer to `i3c_config_t`.

`void I3C_Init(I3C_Type *base, const i3c_config_t *config, uint32_t sourceClock_Hz)`

Initializes the I3C peripheral. This function enables the peripheral clock and initializes the I3C peripheral as described by the user provided configuration. This will initialize both the master peripheral and slave peripheral so that I3C module could work as pure master, pure slave or secondary master, etc. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.
- `config` – User provided peripheral configuration. Use `I3C_GetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`struct _i3c_config`

#include <fsl_i3c.h> Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i3c_master_enable_t enableMaster

Enable master mode.

`bool` disableTimeout

Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep

High keeper mode setting.

`bool` enableOpenDrainStop

Whether to emit open-drain speed STOP.

`bool` enableOpenDrainHigh

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate_Hz

Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay

I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay

I3C SCL delay after Repeated START.

`uint8_t` masterDynamicAddress

Main master dynamic address configuration.

`uint32_t` maxWriteLength

Maximum write length.

`uint32_t` `maxReadLength`
Maximum read length.

`bool` `enableSlave`
Whether to enable slave.

`uint8_t` `staticAddr`
Static address.

`uint16_t` `vendorID`
Device vendor ID(manufacture ID).

`uint32_t` `partNumber`
Device part number info

`uint8_t` `dcr`
Device characteristics register information.

`uint8_t` `bcr`
Bus characteristics register information.

`uint8_t` `hdrMode`
Support hdr mode, could be OR logic in enumeration:`i3c_hdr_mode_t`.

`bool` `nakAllRequest`
Whether to reply NAK to all requests except broadcast CCC.

`bool` `ignoreS0S1Error`
Whether to ignore S0/S1 error in SDR mode.

`bool` `offline`
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

`bool` `matchSlaveStartStop`
Whether to assert start/stop status only the time slave is addressed.

2.31 I3C Master Driver

`void` `I3C_MasterGetDefaultConfig`(`i3c_master_config_t` *`masterConfig`)

Provides a default configuration for the I3C master peripheral.

This function provides the following default configuration for the I3C master peripheral:

```
masterConfig->enableMaster      = kI3C_MasterOn;
masterConfig->disableTimeout    = false;
masterConfig->hKeep             = kI3C_MasterHighKeeperNone;
masterConfig->enableOpenDrainStop = true;
masterConfig->enableOpenDrainHigh = true;
masterConfig->baudRate_Hz       = 100000U;
masterConfig->busType            = kI3C_TypeI2C;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I3C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_master_config_t`.


```
void I3C_MasterInit(I3C_Type *base, const i3c_master_config_t *masterConfig, uint32_t
                    sourceClock_Hz)
```

Initializes the I3C master peripheral.

This function enables the peripheral clock and initializes the I3C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I3C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

```
void I3C_MasterDeinit(I3C_Type *base)
```

Deinitializes the I3C master peripheral.

This function disables the I3C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterCheckAndClearError(I3C_Type *base, uint32_t status)
```

```
status_t I3C_MasterWaitForCtrlDone(I3C_Type *base, bool waitIdle)
```

```
status_t I3C_CheckForBusyBus(I3C_Type *base)
```

```
static inline void I3C_MasterEnable(I3C_Type *base, i3c_master_enable_t enable)
```

Set I3C module master mode.

Parameters

- `base` – The I3C peripheral base address.
- `enable` – Enable master mode.

```
void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)
```

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t
                  slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I3C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I3C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `slowClock_Hz` – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If `FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH` defines as 1, this parameter is useless.

`void I3C_SlaveDeinit(I3C_Type *base)`

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- `base` – The I3C peripheral base address.

`static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)`

Enable/Disable Slave.

Parameters

- `base` – The I3C peripheral base address.
- `isEnabled` – Enable or disable.

`static inline uint32_t I3C_MasterGetStatusFlags(I3C_Type *base)`

Gets the I3C master status flags.

A bit mask with the state of all I3C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_master_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

`static inline void I3C_MasterClearStatusFlags(I3C_Type *base, uint32_t statusMask)`

Clears the I3C master status flag state.

The following status register flags can be cleared:

- `kI3C_MasterSlaveStartFlag`
- `kI3C_MasterControlDoneFlag`
- `kI3C_MasterCompleteFlag`
- `kI3C_MasterArbitrationWonFlag`
- `kI3C_MasterSlave2MasterFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_master_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

```
static inline uint32_t I3C_MasterGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C master error status flags.

A bit mask with the state of all I3C master error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_master_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_MasterClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C master error status flag state.

See also:

`_i3c_master_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_master_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

```
i3c_master_state_t I3C_MasterGetState(I3C_Type *base)
```

Gets the I3C master state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C master state.

```
static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)
```

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

```
static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

`i3c_slave_activity_state_t` `I3C_SlaveGetActivityState(I3C_Type *base)`

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

`status_t` `I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)`

`static inline void` `I3C_MasterEnableInterrupts(I3C_Type *base, uint32_t interruptMask)`

Enables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

`static inline void` `I3C_MasterDisableInterrupts(I3C_Type *base, uint32_t interruptMask)`

Disables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

`static inline uint32_t` `I3C_MasterGetEnabledInterrupts(I3C_Type *base)`

Returns the set of currently enabled I3C master interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

`static inline uint32_t` `I3C_MasterGetPendingInterrupts(I3C_Type *base)`

Returns the set of pending I3C master interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of pending interrupts.

static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_MasterEnableDMA(I3C_Type *base, bool enableTx, bool enableRx,  
                                       uint32_t width)
```

Enables or disables I3C master DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

```
static inline uint32_t I3C_MasterGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C master transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Transmit Data Register address.

```
static inline uint32_t I3C_MasterGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C master receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Receive Data Register address.

```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t  
                                       width)
```

Enables or disables I3C slave DMA requests.

Parameters

- base – The I3C peripheral base address.

- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.
- `width` – DMA read/write unit in bytes.

static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave receive data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

static inline void I3C_MasterSetWatermarks(I3C_Type *base, *i3c_tx_trigger_level_t* txLvl,
i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)

Sets the watermarks for I3C master FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.
- `rxLvl` – Receive FIFO watermark level. The `kI3C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

static inline void I3C_MasterGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)

Gets the current number of bytes in the I3C master FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

static inline void I3C_SlaveSetWatermarks(I3C_Type *base, *i3c_tx_trigger_level_t* txLvl,
i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)

Sets the watermarks for I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.
- `rxLvl` – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
void I3C_MasterSetBaudRate(I3C_Type *base, const i3c_baudrate_hz_t *baudRate_Hz, uint32_t  
sourceClock_Hz)
```

Sets the I3C bus frequency for master transactions.

The I3C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

- `base` – The I3C peripheral base address.
- `baudRate_Hz` – Pointer to structure of requested bus frequency in Hertz.
- `sourceClock_Hz` – I3C functional clock frequency in Hertz.

```
static inline bool I3C_MasterGetBusIdleState(I3C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

```
status_t I3C_MasterStartWithRxSize(I3C_Type *base, i3c_bus_type_t type, uint8_t address,  
i3c_direction_t dir, uint8_t rxSize)
```

Sends a START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the `address` parameter. Note that this function

does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- `rxSize` – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.

status_t I3C_MasterStart(I3C_Type *base, i3c_bus_type_t type, uint8_t address, i3c_direction_t dir)

Sends a START signal and slave address on the I2C/I3C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.

status_t I3C_MasterRepeatedStartWithRxSize(I3C_Type *base, i3c_bus_type_t type, uint8_t address, i3c_direction_t dir, uint8_t rxSize)

Sends a repeated START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `I3C_MasterStart()`, it also sends the specified 7-bit address. Call this API also configures the read terminate size for the following read transfer. For example, set the `rxSize = 2`, the following read transfer will be terminated after two bytes of data received. Write transfer will not be affected by the `rxSize` configuration.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- `rxSize` – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
static inline status_t I3C_MasterRepeatedStart(I3C_Type *base, i3c_bus_type_t type, uint8_t address, i3c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C/I3C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `I3C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
status_t I3C_MasterSend(I3C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)
```

Performs a polling send transfer on the I2C/I3C bus.

Sends up to `txSize` number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_I3C_Nak`.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.
- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.

- `kStatus_I3C_Nak` – The slave device sent a NAK in response to an address.
- `kStatus_I3C_WriteAbort` – The slave device sent a NAK in response to a write.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_WriteFifoError` – Write to M/SWDATAB register when FIFO full.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

status_t I3C_MasterReceive(I3C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transfer on the I2C/I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.
- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Term` – The master terminates slave read.
- `kStatus_I3C_HdrParityError` – Parity error from DDR read.
- `kStatus_I3C_CrcError` – CRC error from DDR read.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_ReadFifoError` – Read from M/SRDATAB register when FIFO empty.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

status_t I3C_MasterStop(I3C_Type *base)

Sends a STOP signal on the I2C/I3C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`void I3C_MasterEmitRequest(I3C_Type *base, i3c_bus_request_t masterReq)`
 I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `masterReq` – I3C master request of type `i3c_bus_request_t`

`static inline void I3C_MasterEmitIBIResponse(I3C_Type *base, i3c_ibi_response_t ibiResponse)`
 I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `ibiResponse` – I3C master emit IBI response of type `i3c_ibi_response_t`

`void I3C_MasterRegisterIBI(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
 I3C master register IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to ibi rule description of type `i3c_register_ibi_addr_t`

`void I3C_MasterGetIBIRules(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
 I3C master get IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to store the read out ibi rule description.

`i3c_ibi_type_t I3C_GetIBIType(I3C_Type *base)`
 I3C master get IBI Type.

Parameters

- `base` – The I3C peripheral base address.

Return values

`i3c_ibi_type_t` – Type of `i3c_ibi_type_t`.

`static inline uint8_t I3C_GetIBIAddress(I3C_Type *base)`
 I3C master get IBI Address.

Parameters

- `base` – The I3C peripheral base address.

Return values

The – 8-bit IBI address.

`status_t I3C_MasterProcessDAASpecifiedBaudrate(I3C_Type *base, uint8_t *addressList, uint32_t count, i3c_master_daa_baudrate_t *daaBaudRate)`

Performs a DAA in the i3c bus with specified temporary baud rate.

Parameters

- `base` – The I3C peripheral base address.
- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list.

- `daaBaudRate` – The temporary baud rate in DAA process, NULL for using initial setting. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
static inline status_t I3C_MasterProcessDAA(I3C_Type *base, uint8_t *addressList, uint32_t count)
```

Performs a DAA in the i3c bus.

Parameters

- `base` – The I3C peripheral base address.
- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
i3c_device_info_t *I3C_MasterGetDeviceListAfterDAA(I3C_Type *base, uint8_t *count)
```

Get device information list after DAA process is done.

Parameters

- `base` – The I3C peripheral base address.
- `count` – **[out]** The pointer to store the available device count.

Returns

Pointer to the `i3c_device_info_t` array.

```
void I3C_MasterClearDeviceCount(I3C_Type *base)
```

Clear the global device count which represents current devices number on the bus. When user resets all dynamic addresses on the bus, should call this API.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterTransferBlocking(I3C_Type *base, i3c_master_transfer_t *transfer)
```

Performs a master polling transfer on the I2C/I3C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- `base` – The I3C peripheral base address.
- `transfer` – Pointer to the transfer structure.

Return values

- `kStatus_I3C_Success` – Data was received successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_IBIWon` – The I3C slave event IBI or MR or HJ won the arbitration on a header address.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Nak` – The slave device sent a NAK in response to an address.
- `kStatus_I3C_WriteAbort` – The slave device sent a NAK in response to a write.
- `kStatus_I3C_Term` – The master terminates slave read.
- `kStatus_I3C_HdrParityError` – Parity error from DDR read.
- `kStatus_I3C_CrcError` – CRC error from DDR read.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_ReadFifoError` – Read from M/SRDATAB register when FIFO empty.
- `kStatus_I3C_WriteFifoError` – Write to M/SWDATAB register when FIFO full.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_MasterTransferCreateHandle(I3C_Type *base, i3c_master_handle_t *handle, const i3c_master_transfer_callback_t *callback, void *userData)`

Creates a new handle for the I3C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The I3C peripheral base address.
- handle – **[out]** Pointer to the I3C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I3C_MasterTransferNonBlocking(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C/I3C bus.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_I3C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t I3C_MasterTransferGetCount(I3C_Type *base, *i3c_master_handle_t* *handle, *size_t* *count)

Returns number of bytes transferred so far.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void I3C_MasterTransferAbort(I3C_Type *base, *i3c_master_handle_t* *handle)

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I3C peripheral's IRQ priority.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.

Return values

- kStatus_Success – A transaction was successfully aborted.
- kStatus_I3C_Idle – There is not a non-blocking transaction currently in progress.

void I3C_MasterTransferHandleIRQ(I3C_Type *base, void *intHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to the I3C master driver handle.

enum _i3c_master_flags

I3C master peripheral flags.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterBetweenFlag

Between messages/DAAs flag

enumerator kI3C_MasterNackDetectFlag

NACK detected flag

enumerator kI3C_MasterSlaveStartFlag

Slave request start flag

enumerator kI3C_MasterControlDoneFlag

Master request complete flag

enumerator kI3C_MasterCompleteFlag

Transfer complete flag

enumerator kI3C_MasterRxReadyFlag

Rx data ready in Rx buffer flag

enumerator kI3C_MasterTxReadyFlag

Tx buffer ready for Tx data flag

enumerator kI3C_MasterArbitrationWonFlag

Header address won arbitration flag

enumerator kI3C_MasterErrorFlag

Error occurred flag

enumerator kI3C_MasterSlave2MasterFlag

Switch from slave to master flag

enumerator kI3C_MasterClearFlags

enum _i3c_master_error_flags

I3C master error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterErrorNackFlag

Slave NACKed the last address

enumerator kI3C_MasterErrorWriteAbortFlag

Slave NACKed the write data

enumerator kI3C_MasterErrorParityFlag

Parity error from DDR read

enumerator kI3C_MasterErrorCrcFlag

CRC error from DDR read

enumerator kI3C_MasterErrorReadFlag

Read from MRDATAB register when FIFO empty

enumerator kI3C_MasterErrorWriteFlag

Write to MWDATAB register when FIFO full

enumerator kI3C_MasterErrorMsgFlag

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kI3C_MasterErrorInvalidReqFlag

Invalid use of request

enumerator kI3C_MasterErrorTimeoutFlag

The module has stalled too long in a frame

enumerator kI3C_MasterAllErrorFlags

All error flags

enum _i3c_master_state

I3C working master state.

Values:

enumerator kI3C_MasterStateIdle

Bus stopped.

enumerator kI3C_MasterStateSlvReq

Bus stopped but slave holding SDA low.

enumerator kI3C_MasterStateMsgSdr

In SDR Message mode from using MWMSG_SDR.

enumerator kI3C_MasterStateNormAct

In normal active SDR mode.

enumerator kI3C_MasterStateDdr

In DDR Message mode.

enumerator kI3C_MasterStateDaa

In ENTDAAs mode.

enumerator kI3C_MasterStateIbiAck

Waiting on IBI ACK/NACK decision.

enumerator kI3C_MasterStateIbiRcv

Receiving IBI.

enum _i3c_master_enable

I3C master enable configuration.

Values:

enumerator kI3C_MasterOff

Master off.

enumerator kI3C_MasterOn

Master on.

enumerator kI3C_MasterCapable

Master capable.

enum _i3c_master_hkeep

I3C high keeper configuration.

Values:

enumerator kI3C_MasterHighKeeperNone

Use PUR to hold SCL high.

enumerator kI3C_MasterHighKeeperWiredIn

Use pin_HK controls.

enumerator kI3C_MasterPassiveSDA

Hi-Z for Bus Free and hold SDA.

enumerator kI3C_MasterPassiveSDASCL

Hi-Z both for Bus Free, and can Hi-Z SDA for hold.

enum _i3c_bus_request

Emits the requested operation when doing in pieces vs. by message.

Values:

enumerator kI3C_RequestNone

No request.

enumerator kI3C_RequestEmitStartAddr

Request to emit start and address on bus.

enumerator kI3C_RequestEmitStop

Request to emit stop on bus.

enumerator kI3C_RequestIbiAckNack

Manual IBI ACK or NACK.

enumerator kI3C_RequestProcessDAA

Process DAA.

enumerator kI3C_RequestForceExit

Request to force exit.

enumerator kI3C_RequestAutoIbi

Hold in stopped state, but Auto-emit START,7E.

enum __i3c_bus_type

Bus type with EmitStartAddr.

Values:

enumerator kI3C_TypeI3CSdr

SDR mode of I3C.

enumerator kI3C_TypeI2C

Standard i2c protocol.

enumerator kI3C_TypeI3CDdr

HDR-DDR mode of I3C.

enum __i3c_ibi_response

IBI response.

Values:

enumerator kI3C_IbiRespAck

ACK with no mandatory byte.

enumerator kI3C_IbiRespNack

NACK.

enumerator kI3C_IbiRespAckMandatory

ACK with mandatory byte.

enumerator kI3C_IbiRespManual

Reserved.

enum __i3c_ibi_type

IBI type.

Values:

enumerator kI3C_IbiNormal

In-band interrupt.

enumerator kI3C_IbiHotJoin

slave hot join.

enumerator kI3C_IbiMasterRequest

slave master ship request.

enum __i3c_ibi_state

IBI state.

Values:

enumerator kI3C_IbiReady

In-band interrupt ready state, ready for user to handle.

enumerator kI3C_IbiDataBuffNeed

In-band interrupt need data buffer for data receive.

enumerator kI3C_IbiAckNackPending

In-band interrupt Ack/Nack pending for decision.

enum _i3c_direction

Direction of master and slave transfers.

Values:

enumerator kI3C_Write

Master transmit.

enumerator kI3C_Read

Master receive.

enum _i3c_tx_trigger_level

Watermark of TX int/dma trigger level.

Values:

enumerator kI3C_TxTriggerOnEmpty

Trigger on empty.

enumerator kI3C_TxTriggerUntilOneQuarterOrLess

Trigger on 1/4 full or less.

enumerator kI3C_TxTriggerUntilOneHalfOrLess

Trigger on 1/2 full or less.

enumerator kI3C_TxTriggerUntilOneLessThanFull

Trigger on 1 less than full or less.

enum _i3c_rx_trigger_level

Watermark of RX int/dma trigger level.

Values:

enumerator kI3C_RxTriggerOnNotEmpty

Trigger on not empty.

enumerator kI3C_RxTriggerUntilOneQuarterOrMore

Trigger on 1/4 full or more.

enumerator kI3C_RxTriggerUntilOneHalfOrMore

Trigger on 1/2 full or more.

enumerator kI3C_RxTriggerUntilThreeQuarterOrMore

Trigger on 3/4 full or more.

enum _i3c_rx_term_ops

I3C master read termination operations.

Values:

enumerator kI3C_RxTermDisable

Master doesn't terminate read, used for CCC transfer.

enumerator kI3C_RxAutoTerm

Master auto terminate read after receiving specified bytes(<=255).

enumerator kI3C_RxTermLastByte

Master terminates read at any time after START, no length limitation.

enum _i3c_start_scl_delay

I3C start SCL delay options.

Values:

enumerator kI3C_NoDelay

No delay.

enumerator kI3C_IncreaseSclHalfPeriod

Increases SCL clock period by 1/2.

enumerator kI3C_IncreaseSclOnePeriod

Increases SCL clock period by 1.

enumerator kI3C_IncreaseSclOneAndHalfPeriod

Increases SCL clock period by 1 1/2

enum _i3c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_i3c_master_transfer::flags` field.

Values:

enumerator kI3C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kI3C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kI3C_TransferRepeatedStartFlag

Send a repeated start condition

enumerator kI3C_TransferNoStopFlag

Don't send a stop condition.

enumerator kI3C_TransferWordsFlag

Transfer in words, else transfer in bytes.

enumerator kI3C_TransferDisableRxTermFlag

Disable Rx termination. Note: It's for I3C CCC transfer.

enumerator kI3C_TransferRxAutoTermFlag

Set Rx auto-termination. Note: It's adaptive based on Rx size(<=255 bytes) except in I3C_MasterReceive.

enumerator kI3C_TransferStartWithBroadcastAddr

Start transfer with 0x7E, then read/write data with device address.

typedef enum _i3c_master_state i3c_master_state_t

I3C working master state.

typedef enum _i3c_master_enable i3c_master_enable_t

I3C master enable configuration.

typedef enum _i3c_master_hkeep i3c_master_hkeep_t

I3C high keeper configuration.

typedef enum _i3c_bus_request i3c_bus_request_t

Emits the requested operation when doing in pieces vs. by message.

typedef enum _i3c_bus_type i3c_bus_type_t

Bus type with EmitStartAddr.

```
typedef enum _i3c_ibi_response i3c_ibi_response_t
    IBI response.
```

```
typedef enum _i3c_ibi_type i3c_ibi_type_t
    IBI type.
```

```
typedef enum _i3c_ibi_state i3c_ibi_state_t
    IBI state.
```

```
typedef enum _i3c_direction i3c_direction_t
    Direction of master and slave transfers.
```

```
typedef enum _i3c_tx_trigger_level i3c_tx_trigger_level_t
    Watermark of TX int/dma trigger level.
```

```
typedef enum _i3c_rx_trigger_level i3c_rx_trigger_level_t
    Watermark of RX int/dma trigger level.
```

```
typedef enum _i3c_rx_term_ops i3c_rx_term_ops_t
    I3C master read termination operations.
```

```
typedef enum _i3c_start_scl_delay i3c_start_scl_delay_t
    I3C start SCL delay options.
```

```
typedef struct _i3c_register_ibi_addr i3c_register_ibi_addr_t
    Structure with setting master IBI rules and slave registry.
```

```
typedef struct _i3c_baudrate i3c_baudrate_hz_t
    Structure with I3C baudrate settings.
```

```
typedef struct _i3c_master_daa_baudrate i3c_master_daa_baudrate_t
    I3C DAA baud rate configuration.
```

```
typedef struct _i3c_master_config i3c_master_config_t
    Structure with settings to initialize the I3C master module.
```

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_MasterGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef struct _i3c_master_transfer i3c_master_transfer_t
```

```
typedef struct _i3c_master_handle i3c_master_handle_t
```

```
typedef struct _i3c_master_transfer_callback i3c_master_transfer_callback_t
    i3c master callback functions.
```

```
typedef void (*i3c_master_isr_t)(I3C_Type *base, void *handle)
    Typedef for master interrupt handler.
```

```
struct _i3c_register_ibi_addr
    #include <fsl_i3c.h> Structure with setting master IBI rules and slave registry.
```

Public Members

```
uint8_t address[5]
    Address array for registry.
```

```
bool i3cFastStart
    Allow the START header to run as push-pull speed if all dynamic addresses take MSB 0.
```

bool ibiHasPayload

Whether the address array has mandatory IBI byte.

struct __i3c__baudrate

#include <fsl_i3c.h> Structure with I3C baudrate settings.

Public Members

uint32_t i2cBaud

Desired I2C baud rate in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct __i3c__master__daa__baudrate

#include <fsl_i3c.h> I3C DAA baud rate configuration.

Public Members

uint32_t sourceClock_Hz

FCLK, function clock in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct __i3c__master__config

#include <fsl_i3c.h> Structure with settings to initialize the I3C master module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i3c_master_enable_t enableMaster

Enable master mode.

bool disableTimeout

Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep

High keeper mode setting.

bool enableOpenDrainStop

Whether to emit open-drain speed STOP.

bool enableOpenDrainHigh

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate_Hz

Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay

I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay

I3C SCL delay after Repeated START.

struct *_i3c_master_transfer_callback*

#include <fsl_i3c.h> i3c master callback functions.

Public Members

void (*slave2Master)(I3C_Type *base, void *userData)

Transfer complete callback

void (*ibiCallback)(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_ibi_type_t* ibiType, *i3c_ibi_state_t* ibiState)

IBI event callback

void (*transferComplete)(I3C_Type *base, *i3c_master_handle_t* *handle, *status_t* completionStatus, void *userData)

Transfer complete callback

struct *_i3c_master_transfer*

#include <fsl_i3c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I3C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration *_i3c_master_transfer_flags* for available options. Set to 0 or *kI3C_TransferDefaultFlag* for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i3c_direction_t direction

Either *kI3C_Read* or *kI3C_Write*.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

i3c_bus_type_t busType

bus type.

i3c_ibi_response_t ibiResponse

ibi response during transfer.

```
struct _i3c_master_handle
    #include <fsl_i3c.h> Driver handle for master non-blocking APIs.
```

Note: The contents of this structure are private and subject to change.

Public Members

`uint8_t` state
Transfer state machine current state.

`uint32_t` remainingBytes
Remaining byte count in current state.

`i3c_rx_term_ops_t` rxTermOps
Read termination operation.

`i3c_master_transfer_t` transfer
Copy of the current transfer info.

`uint8_t` ibiAddress
Slave address which request IBI.

`uint8_t *ibiBuff`
Pointer to IBI buffer to keep ibi bytes.

`size_t` ibiPayloadSize
IBI payload size.

`i3c_ibi_type_t` ibiType
IBI type.

`i3c_master_transfer_callback_t` callback
Callback functions pointer.

`void *userData`
Application data passed to callback.

2.32 I3C Slave Driver

`void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)`
Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableSlave = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t  
slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- base – The I3C peripheral base address.
- slaveConfig – User provided peripheral configuration. Use I3C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
- slowClock_Hz – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH defines as 1, this parameter is useless.

```
void I3C_SlaveDeinit(I3C_Type *base)
```

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- base – The I3C peripheral base address.

```
static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)
```

Enable/Disable Slave.

Parameters

- base – The I3C peripheral base address.
- isEnabled – Enable or disable.

```
static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)
```

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

[_i3c_slave_flags](#)

Parameters

- base – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

`static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)`

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

`static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)`

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

`i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)`

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

`status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)`

```
static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C slave interrupt requests.

Parameters

- `base` – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)
```

Enables or disables I3C slave DMA requests.

Parameters

- `base` – The I3C peripheral base address.
- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.
- `width` – DMA read/write unit in bytes.

```
static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

```
static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave receive data register address for DMA transfer.

Parameters

- `base` – The I3C peripheral base address.
- `width` – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

```
static inline void I3C_SlaveSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl, i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)
```

Sets the watermarks for I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.

- `rxLvl` – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

`static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)`

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_SlaveTransferCreateHandle(I3C_Type *base, i3c_slave_handle_t *handle, i3c_slave_transfer_callback_t callback, void *userData)`

Creates a new handle for the I3C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_SlaveTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C slave driver handle.

- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_SlaveTransferNonBlocking(I3C_Type *base, i3c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling `I2C_SlaveInit()` and `I3C_SlaveTransferCreateHandle()` to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to `I3C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i3c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI3C_SlaveTransmitEvent` and `kI3C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI3C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i3c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI3C_SlaveAllEvents` to enable all events.

Return values

- `kStatus__Success` – Slave transfers were successfully started.
- `kStatus_I3C_Busy` – Slave transfers have already been started on this handle.

`status_t I3C_SlaveTransferGetCount(I3C_Type *base, i3c_slave_handle_t *handle, size_t *count)`

Gets the slave transfer status during a non-blocking transfer.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure.
- `count` – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- `kStatus__Success` –
- `kStatus_NoTransferInProgress` –

`void I3C_SlaveTransferAbort(I3C_Type *base, i3c_slave_handle_t *handle)`

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- `base` – The I3C peripheral base address.

- handle – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

Return values

- `kStatus__Success` –
- `kStatus_I3C_Idle` –

`void I3C_SlaveTransferHandleIRQ(I3C_Type *base, void *intHandle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`enum _i3c_slave_flags`

I3C slave peripheral flags.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator `kI3C_SlaveNotStopFlag`

Slave status not stop flag

enumerator `kI3C_SlaveMessageFlag`

Slave status message, indicating slave is listening to the bus traffic or responding

enumerator kI3C_SlaveRequiredReadFlag
Slave status required, either is master doing SDR read from slave, or is IBI pushing out.

enumerator kI3C_SlaveRequiredWriteFlag
Slave status request write, master is doing SDR write to slave, except slave in ENTDAAMode

enumerator kI3C_SlaveBusDAAFlag
I3C bus is in ENTDAAMode

enumerator kI3C_SlaveBusHDRModeFlag
I3C bus is in HDR mode

enumerator kI3C_SlaveBusStartFlag
Start/Re-start event is seen since the bus was last cleared

enumerator kI3C_SlaveMatchedFlag
Slave address(dynamic/static) matched since last cleared

enumerator kI3C_SlaveBusStopFlag
Stop event is seen since the bus was last cleared

enumerator kI3C_SlaveRxReadyFlag
Rx data ready in rx buffer flag

enumerator kI3C_SlaveTxReadyFlag
Tx buffer ready for Tx data flag

enumerator kI3C_SlaveDynamicAddrChangedFlag
Slave dynamic address has been assigned, re-assigned, or lost

enumerator kI3C_SlaveReceivedCCCFlag
Slave received Common command code

enumerator kI3C_SlaveErrorFlag
Error occurred flag

enumerator kI3C_SlaveHDRCommandMatchFlag
High data rate command match

enumerator kI3C_SlaveCCCHandledFlag
Slave received Common command code is handled by I3C module

enumerator kI3C_SlaveEventSentFlag
Slave IBI/P2P/MR/HJ event has been sent

enumerator kI3C_SlaveIbiDisableFlag
Slave in band interrupt is disabled.

enumerator kI3C_SlaveMasterRequestDisabledFlag
Slave master request is disabled.

enumerator kI3C_SlaveHotJoinDisabledFlag
Slave Hot-Join is disabled.

enumerator kI3C_SlaveClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kI3C_SlaveAllIrqFlags

enum _i3c_slave_error_flags

I3C slave error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_SlaveErrorOvrerrunFlag

Slave internal from-bus buffer/FIFO overrun.

enumerator kI3C_SlaveErrorUnderrunFlag

Slave internal to-bus buffer/FIFO underrun

enumerator kI3C_SlaveErrorUnderrunNakFlag

Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kI3C_SlaveErrorTermFlag

Terminate error from master

enumerator kI3C_SlaveErrorInvalidStartFlag

Slave invalid start flag

enumerator kI3C_SlaveErrorSdrParityFlag

SDR parity error

enumerator kI3C_SlaveErrorHdrParityFlag

HDR parity error

enumerator kI3C_SlaveErrorHdrCRCFlag

HDR-DDR CRC error

enumerator kI3C_SlaveErrorS0S1Flag

S0 or S1 error

enumerator kI3C_SlaveErrorOverreadFlag

Over-read error

enumerator kI3C_SlaveErrorOverwriteFlag

Over-write error

enum _i3c_slave_event

I3C slave.event.

Values:

enumerator kI3C_SlaveEventNormal

Normal mode.

enumerator kI3C_SlaveEventIBI

In band interrupt event.

enumerator kI3C_SlaveEventMasterReq

Master request event.

enumerator kI3C_SlaveEventHotJoinReq

Hot-join event.

enum _i3c_slave_activity_state

I3C slave.activity state.

Values:

enumerator kI3C_SlaveNoLatency

Normal bus operation

enumerator kI3C_SlaveLatency1Ms

1ms of latency.

enumerator kI3C_SlaveLatency100Ms

100ms of latency.

enumerator kI3C_SlaveLatency10S

10s latency.

enum _i3c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kI3C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kI3C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kI3C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveRequiredTransmitEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveStartEvent

A start/repeated start was detected.

enumerator kI3C_SlaveHDRCommandMatchEvent

Slave Match HDR Command.

enumerator kI3C_SlaveCompletionEvent

A stop was detected, completing the transfer.

enumerator kI3C_SlaveRequestSentEvent

Slave request event sent.

enumerator kI3C_SlaveReceivedCCCEvent

Slave received CCC event, need to handle by application.

enumerator kI3C_SlaveAllEvents

Bit mask of all available events.

typedef enum _i3c_slave_event i3c_slave_event_t

I3C slave.event.

typedef enum _i3c_slave_activity_state i3c_slave_activity_state_t

I3C slave.activity state.

```
typedef struct _i3c_slave_config i3c_slave_config_t
```

Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _i3c_slave_transfer_event i3c_slave_transfer_event_t
```

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

```
typedef struct _i3c_slave_transfer i3c_slave_transfer_t
```

I3C slave transfer structure.

```
typedef struct _i3c_slave_handle i3c_slave_handle_t
```

```
typedef void (*i3c_slave_transfer_callback_t)(I3C_Type *base, i3c_slave_transfer_t *transfer, void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the I3C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the I3C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef void (*i3c_slave_isr_t)(I3C_Type *base, void *handle)
```

Typedef for slave interrupt handler.

```
struct _i3c_slave_config
```

#include <fsl_i3c.h> Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableSlave

Whether to enable slave.

uint8_t staticAddr

Static address.

`uint16_t vendorID`
Device vendor ID(manufacture ID).

`uint32_t partNumber`
Device part number info

`uint8_t dcr`
Device characteristics register information.

`uint8_t bcr`
Bus characteristics register information.

`uint8_t hdrMode`
Support hdr mode, could be OR logic in enumeration:`i3c_hdr_mode_t`.

`bool nakAllRequest`
Whether to reply NAK to all requests except broadcast CCC.

`bool ignoreS0S1Error`
Whether to ignore S0/S1 error in SDR mode.

`bool offline`
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

`bool matchSlaveStartStop`
Whether to assert start/stop status only the time slave is addressed.

`uint32_t maxWriteLength`
Maximum write length.

`uint32_t maxReadLength`
Maximum read length.

`struct _i3c_slave_transfer`
#include <fsl_i3c.h> I3C slave transfer structure.

Public Members

`uint32_t event`
Reason the callback is being invoked.

`uint8_t *txData`
Transfer buffer

`size_t txDataSize`
Transfer size

`uint8_t *rxData`
Transfer buffer

`size_t rxDataSize`
Transfer size

`status_t completionStatus`
Success or error code describing how the transfer completed. Only applies for `ki3c_slave_completion_event`.

`size_t transferredCount`
Number of bytes actually transferred since start or last repeated start.

```
struct _i3c_slave_handle
#include <fsl_i3c.h> I3C slave handle structure.
```

Note: The contents of this structure are private and subject to change.

Public Members

i3c_slave_transfer_t transfer
I3C slave transfer copy.

bool isBusy
Whether transfer is busy.

bool wasTransmit
Whether the last transfer was a transmit.

uint32_t eventMask
Mask of enabled events.

uint32_t transferredCount
Count of bytes transferred.

i3c_slave_transfer_callback_t callback
Callback function called at transfer event.

void *userData
Callback parameter passed to callback.

uint8_t txFifoSize
Tx Fifo size

2.33 Iomuxc_driver

```
static inline void IOMUXC_SetPinMux(uint32_t muxRegister, uint32_t muxMode, uint32_t
inputRegister, uint32_t inputDaisy, uint32_t
configRegister, uint32_t inputInv)
```

Sets the IOMUXC pin mux mode.

This is an example to set the PTA2 as the lpuart0_tx:

```
IOMUXC_SetPinMux(IOMUXC_PTA2_LPUART0_TX, 0);
```

This is an example to set the PTC13 as i2s1_txd3:

```
IOMUXC_SetPinMux(IOMUXC_PTC13_I2S1_TXD3, 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.
- inputInv – The pad->module input inversion.

```
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_PTA7_LPUART1_RX:

```
IOMUXC_SetPinConfig(IOMUXC_PTA7_LPUART1_RX,
                    IOMUXC_PCR_PE_MASK |
                    IOMUXC_PCR_PS_MASK)
```

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.

FSL_IOMUXC_DRIVER_VERSION

IOMUXC driver version 2.0.0.

IOMUXC_PTA0_CMP1_IN2

IOMUXC_PTA0_PTA0

IOMUXC_PTA0_LPSPI0_PCS1

IOMUXC_PTA0_LPUART0_CTS_B

IOMUXC_PTA0_LPI2C0_SCL

IOMUXC_PTA0_TPM0_CLKIN

IOMUXC_PTA0_I2S0_RX_BCLK

IOMUXC_PTA0_LPTMR0_ALT1

IOMUXC_PTA0_WUU0_P0

IOMUXC_PTA1_CMP1_IN3

IOMUXC_PTA1_PTA1

IOMUXC_PTA1_LPSPI0_PCS2

IOMUXC_PTA1_LPUART0_RTS_B

IOMUXC_PTA1_LPI2C0_SDA
IOMUXC_PTA1_TPM0_CH0
IOMUXC_PTA1_I2S0_RX_FS
IOMUXC_PTA1_EWM0_OUT_B
IOMUXC_PTA1_RTC_CLKOUT_B
IOMUXC_PTA2_CMP1_IN4
IOMUXC_PTA2_PTA2
IOMUXC_PTA2_LPSP10_PCS3
IOMUXC_PTA2_LPUART0_TX
IOMUXC_PTA2_LPI2C0_HREQ
IOMUXC_PTA2_TPM0_CH1
IOMUXC_PTA2_I2S0_RXD0
IOMUXC_PTA2_WDOG0_RST
IOMUXC_PTA2_LPTMR0_ALT2
IOMUXC_PTA2_RTC_CLKOUT
IOMUXC_PTA3_CMP1_IN5
IOMUXC_PTA3_PTA3
IOMUXC_PTA3_LPUART0_RX
IOMUXC_PTA3_LPI2C1_HREQ
IOMUXC_PTA3_TPM0_CH2
IOMUXC_PTA3_I2S0_RXD1
IOMUXC_PTA3_PMIC0_MODE2
IOMUXC_PTA3_CMP0_OUT
IOMUXC_PTA3_WUU0_P1
IOMUXC_PTA4_CMP0_IN2
IOMUXC_PTA4_PTA4
IOMUXC_PTA4_LPSP10_SIN
IOMUXC_PTA4_LPUART1_CTS_B
IOMUXC_PTA4_LPI2C1_SCL
IOMUXC_PTA4_TPM0_CH3
IOMUXC_PTA4_I2S0_MCLK
IOMUXC_PTA4_EXT_AUD_MCLK0
IOMUXC_PTA4_CAN0_TX

IOMUXC_PTA4_PMIC0_MODE1
IOMUXC_PTA4_LPTMR0_ALT3
IOMUXC_PTA4_WUU0_P2
IOMUXC_PTA5_CMP0_IN3
IOMUXC_PTA5_PTA5
IOMUXC_PTA5_LPSPi0_SOUT
IOMUXC_PTA5_LPUART1_RTS_B
IOMUXC_PTA5_LPI2C1_SDA
IOMUXC_PTA5_TPM0_CH4
IOMUXC_PTA5_I2S0_TX_BCLK
IOMUXC_PTA5_CAN0_RX
IOMUXC_PTA5_LPTMR1_ALT1
IOMUXC_PTA5_RTC_CLKOUT_B
IOMUXC_PTA6_CMP0_IN4
IOMUXC_PTA6_PTA6
IOMUXC_PTA6_LPSPi0_SCK
IOMUXC_PTA6_LPUART1_TX
IOMUXC_PTA6_I3C0_SCL
IOMUXC_PTA6_TPM0_CH5
IOMUXC_PTA6_I2S0_TX_FS
IOMUXC_PTA6_SEC0_TX
IOMUXC_PTA6_RTC_CLKOUT
IOMUXC_PTA6_WUU0_P3
IOMUXC_PTA7_CMP0_IN5
IOMUXC_PTA7_PTA7
IOMUXC_PTA7_LPSPi0_PCS0
IOMUXC_PTA7_LPUART1_RX
IOMUXC_PTA7_I3C0_SDA
IOMUXC_PTA7_TPM1_CLKIN
IOMUXC_PTA7_I2S0_TXD0
IOMUXC_PTA7_SEC0_RX
IOMUXC_PTA7_WDOG1_RST
IOMUXC_PTA7_LPTMR1_ALT2

IOMUXC_PTA7_WUU0_P4
IOMUXC_PTA8_ADC1_CH0A
IOMUXC_PTA8_PTA8
IOMUXC_PTA8_FXIO0_D0
IOMUXC_PTA8_LPSPI1_PCS1
IOMUXC_PTA8_LPUART1_CTS_B
IOMUXC_PTA8_LPI2C0_SCL
IOMUXC_PTA8_TPM1_CH0
IOMUXC_PTA8_I2S0_TXD1
IOMUXC_PTA8_PMIC0_MODE0
IOMUXC_PTA8_LPTMR1_ALT3
IOMUXC_PTA8_WUU0_P5
IOMUXC_PTA9_ADC1_CH0B
IOMUXC_PTA9_PTA9
IOMUXC_PTA9_FXIO0_D1
IOMUXC_PTA9_LPSPI1_PCS2
IOMUXC_PTA9_LPUART1_RTS_B
IOMUXC_PTA9_LPI2C0_SDA
IOMUXC_PTA9_TPM1_CH1
IOMUXC_PTA9_I2S1_RX_BCLK
IOMUXC_PTA9_NMI0_B
IOMUXC_PTA9_WUU0_P6
IOMUXC_PTA10_ADC1_CH1A
IOMUXC_PTA10_PTA10
IOMUXC_PTA10_FXIO0_D2
IOMUXC_PTA10_LPSPI1_PCS3
IOMUXC_PTA10_LPUART1_TX
IOMUXC_PTA10_LPI2C0_HREQ
IOMUXC_PTA10_I3C0_PUR
IOMUXC_PTA10_I2S1_RX_FS
IOMUXC_PTA10_EWM0_IN
IOMUXC_PTA10_WUU0_P7
IOMUXC_PTA11_ADC1_CH1B

IOMUXC_PTA11_PTA11
IOMUXC_PTA11_FXIO0_D3
IOMUXC_PTA11_LPUART1_RX
IOMUXC_PTA11_LPI2C1_HREQ
IOMUXC_PTA11_I3C0_PUR
IOMUXC_PTA11_I2S1_RXD0
IOMUXC_PTA11_WDOG0_RST
IOMUXC_PTA11_LPTMR0_ALT1
IOMUXC_PTA11_WUU0_P8
IOMUXC_PTA12_ADC1_CH2A
IOMUXC_PTA12_PTA12
IOMUXC_PTA12_FXIO0_D4
IOMUXC_PTA12_LPSPI1_SIN
IOMUXC_PTA12_LPUART0_CTS_B
IOMUXC_PTA12_LPI2C1_SCL
IOMUXC_PTA12_I2S1_RXD1
IOMUXC_PTA12_CAN0_TX
IOMUXC_PTA12_PMIC0_SDA
IOMUXC_PTA12_EWM0_IN
IOMUXC_PTA12_WUU0_P9
IOMUXC_PTA13_ADC1_CH2B
IOMUXC_PTA13_PTA13
IOMUXC_PTA13_FXIO0_D5
IOMUXC_PTA13_LPSPI1_SOUT
IOMUXC_PTA13_LPUART0_RTS_B
IOMUXC_PTA13_LPI2C1_SDA
IOMUXC_PTA13_I2S1_MCLK
IOMUXC_PTA13_EXT_AUD_MCLK0
IOMUXC_PTA13_CAN0_RX
IOMUXC_PTA13_PMIC0_SCL
IOMUXC_PTA13_CMP0_OUT
IOMUXC_PTA13_WUU0_P10
IOMUXC_PTA14_ADC1_CH3A

IOMUXC_PTA14_PTA14
IOMUXC_PTA14_FXIO0_D6
IOMUXC_PTA14_LPSPI1_SCK
IOMUXC_PTA14_LPUART0_TX
IOMUXC_PTA14_I3C0_SCL
IOMUXC_PTA14_TPM0_CLKIN
IOMUXC_PTA14_I2S1_TX_BCLK
IOMUXC_PTA14_SEC0_TX
IOMUXC_PTA14_EWM0_OUT_B
IOMUXC_PTA14_WUU0_P11
IOMUXC_PTA15_ADC1_CH3B
IOMUXC_PTA15_PTA15
IOMUXC_PTA15_FXIO0_D7
IOMUXC_PTA15_LPSPI1_PCS0
IOMUXC_PTA15_LPUART0_RX
IOMUXC_PTA15_I3C0_SDA
IOMUXC_PTA15_TPM0_CH0
IOMUXC_PTA15_I2S1_TX_FS
IOMUXC_PTA15_SEC0_RX
IOMUXC_PTA15_CLKOUT0
IOMUXC_PTA15_CMP1_OUT
IOMUXC_PTA15_WUU0_P12
IOMUXC_PTA16_ADC1_CH4A
IOMUXC_PTA16_PTA16
IOMUXC_PTA16_FXIO0_D8
IOMUXC_PTA16_LPSPI1_SIN
IOMUXC_PTA16_LPUART0_CTS_B
IOMUXC_PTA16_LPI2C0_SCL
IOMUXC_PTA16_TPM0_CH1
IOMUXC_PTA16_I2S1_TXD0
IOMUXC_PTA16_CAN0_TX
IOMUXC_PTA16_RTC_CLKOUT_B
IOMUXC_PTA16_WUU0_P13

IOMUXC_PTA17_ADC1_CH4B
IOMUXC_PTA17_PTA17
IOMUXC_PTA17_FXIO0_D9
IOMUXC_PTA17_LPSPI1_SOUT
IOMUXC_PTA17_LPUART0_RTS_B
IOMUXC_PTA17_LPI2C0_SDA
IOMUXC_PTA17_TPM0_CH2
IOMUXC_PTA17_I2S1_TXD1
IOMUXC_PTA17_CAN0_RX
IOMUXC_PTA17_WDOG1_RST
IOMUXC_PTA17_RTC_CLKOUT
IOMUXC_PTA17_WUU0_P14
IOMUXC_PTA18_ADC1_CH5A
IOMUXC_PTA18_PTA18
IOMUXC_PTA18_FXIO0_D10
IOMUXC_PTA18_LPSPI1_SCK
IOMUXC_PTA18_LPUART0_TX
IOMUXC_PTA18_LPI2C0_HREQ
IOMUXC_PTA18_TPM0_CH3
IOMUXC_PTA18_I2S1_TXD2
IOMUXC_PTA18_NMI1_B
IOMUXC_PTA18_WUU0_P15
IOMUXC_PTA19_PTA19
IOMUXC_PTA19_FXIO0_D11
IOMUXC_PTA19_LPSPI1_PCS0
IOMUXC_PTA19_LPUART0_RX
IOMUXC_PTA19_LPI2C1_HREQ
IOMUXC_PTA19_TPM1_CLKIN
IOMUXC_PTA19_I2S1_TXD3
IOMUXC_PTA19_MQS0_LEFT
IOMUXC_PTA19_JTAG0_TRST_B
IOMUXC_PTA19_LPTMR0_ALT2
IOMUXC_PTA20_PTA20

IOMUXC_PTA20_FXIO0_D12
IOMUXC_PTA20_LPSPI0_SIN
IOMUXC_PTA20_LPUART1_CTS_B
IOMUXC_PTA20_LPI2C1_SCL
IOMUXC_PTA20_TPM1_CH0
IOMUXC_PTA20_I2S1_RXD2
IOMUXC_PTA20_JTAG0_TMS_SWD0_DIO
IOMUXC_PTA20_LPTMR0_ALT3
IOMUXC_PTA21_PTA21
IOMUXC_PTA21_FXIO0_D13
IOMUXC_PTA21_LPSPI0_SOUT
IOMUXC_PTA21_LPUART1_RTS_B
IOMUXC_PTA21_LPI2C1_SDA
IOMUXC_PTA21_TPM1_CH1
IOMUXC_PTA21_I2S1_RXD3
IOMUXC_PTA21_MQS0_RIGHT
IOMUXC_PTA21_JTAG0_TDO
IOMUXC_PTA22_PTA22
IOMUXC_PTA22_FXIO0_D14
IOMUXC_PTA22_LPSPI0_SCK
IOMUXC_PTA22_LPUART1_TX
IOMUXC_PTA22_I3C0_SCL
IOMUXC_PTA22_MQS0_LEFT
IOMUXC_PTA22_JTAG0_TDI
IOMUXC_PTA23_PTA23
IOMUXC_PTA23_FXIO0_D15
IOMUXC_PTA23_LPSPI0_PCS0
IOMUXC_PTA23_LPUART1_RX
IOMUXC_PTA23_I3C0_SDA
IOMUXC_PTA23_MQS0_RIGHT
IOMUXC_PTA23_JTAG0_TCLK_SWD0_CLK
IOMUXC_PTA24_ADC1_CH5B
IOMUXC_PTA24_PTA24

IOMUXC_PTA24_I3C0_PUR
IOMUXC_PTA24_EXT_AUD_MCLK0
IOMUXC_PTA24_WDOG2_RST
IOMUXC_PTA24_LPTMR1_ALT1
IOMUXC_PTA24_WUU0_P16
IOMUXC_PTB0_ADC0_CH0A
IOMUXC_PTB0_PTB0
IOMUXC_PTB0_FXIO0_D16
IOMUXC_PTB0_LPSPI2_PCS1
IOMUXC_PTB0_LPUART2_CTS_B
IOMUXC_PTB0_LPI2C2_SCL
IOMUXC_PTB0_TPM2_CLKIN
IOMUXC_PTB0_I2S2_RXD1
IOMUXC_PTB0_MQS0_LEFT
IOMUXC_PTB0_MICFIL0_CLK01
IOMUXC_PTB0_WUU0_P17
IOMUXC_PTB1_ADC0_CH0B
IOMUXC_PTB1_PTB1
IOMUXC_PTB1_FXIO0_D17
IOMUXC_PTB1_LPSPI2_PCS2
IOMUXC_PTB1_LPUART2_RTS_B
IOMUXC_PTB1_LPI2C2_SDA
IOMUXC_PTB1_TPM2_CH0
IOMUXC_PTB1_I2S2_RX_BCLK
IOMUXC_PTB1_MICFIL0_DATA01
IOMUXC_PTB1_WDOG2_RST
IOMUXC_PTB1_WUU0_P18
IOMUXC_PTB2_ADC0_CH1A
IOMUXC_PTB2_PTB2
IOMUXC_PTB2_FXIO0_D18
IOMUXC_PTB2_LPSPI2_PCS3
IOMUXC_PTB2_LPUART2_TX
IOMUXC_PTB2_LPI2C2_HREQ

IOMUXC_PTB2_TPM2_CH1
IOMUXC_PTB2_I2S2_RX_FS
IOMUXC_PTB2_MICFIL0_CLK01
IOMUXC_PTB2_LPTMR0_ALT1
IOMUXC_PTB2_WUU0_P19
IOMUXC_PTB3_ADC0_CH1B
IOMUXC_PTB3_PTB3
IOMUXC_PTB3_FXIO0_D19
IOMUXC_PTB3_LPSPI2_SIN
IOMUXC_PTB3_LPUART2_RX
IOMUXC_PTB3_LPI2C2_HREQ
IOMUXC_PTB3_I3C1_PUR
IOMUXC_PTB3_I2S2_RXD0
IOMUXC_PTB3_MICFIL0_DATA23
IOMUXC_PTB3_LPTMR0_ALT2
IOMUXC_PTB3_WUU0_P20
IOMUXC_PTB4_ADC0_CH2A
IOMUXC_PTB4_PTB4
IOMUXC_PTB4_FXIO0_D20
IOMUXC_PTB4_LPSPI2_SOUT
IOMUXC_PTB4_LPUART2_CTS_B
IOMUXC_PTB4_LPI2C2_SCL
IOMUXC_PTB4_I2S2_TX_BCLK
IOMUXC_PTB4_MICFIL0_CLK01
IOMUXC_PTB4_WDOG0_RST
IOMUXC_PTB4_LPTMR0_ALT3
IOMUXC_PTB4_WUU0_P21
IOMUXC_PTB5_ADC0_CH2B
IOMUXC_PTB5_PTB5
IOMUXC_PTB5_FXIO0_D21
IOMUXC_PTB5_LPSPI2_SCK
IOMUXC_PTB5_LPUART2_RTS_B
IOMUXC_PTB5_LPI2C2_SDA

IOMUXC_PTB5_I2S2_TX_FS
IOMUXC_PTB5_MICFIL0_DATA45
IOMUXC_PTB5_WDOG1_RST
IOMUXC_PTB5_WUU0_P22
IOMUXC_PTB6_ADC0_CH3A
IOMUXC_PTB6_PTB6
IOMUXC_PTB6_FXIO0_D22
IOMUXC_PTB6_LPSPI2_PCS0
IOMUXC_PTB6_LPUART2_TX
IOMUXC_PTB6_I3C1_SCL
IOMUXC_PTB6_I2S2_TXD0
IOMUXC_PTB6_MICFIL0_CLK01
IOMUXC_PTB6_WUU0_P23
IOMUXC_PTB7_ADC0_CH3B
IOMUXC_PTB7_PTB7
IOMUXC_PTB7_FXIO0_D23
IOMUXC_PTB7_LPSPI3_PCS1
IOMUXC_PTB7_LPUART2_RX
IOMUXC_PTB7_I3C1_SDA
IOMUXC_PTB7_TPM3_CLKIN
IOMUXC_PTB7_I2S2_MCLK
IOMUXC_PTB7_EXT_AUD_MCLK1
IOMUXC_PTB7_MICFIL0_DATA67
IOMUXC_PTB7_PMIC0_MODE2
IOMUXC_PTB8_ADC0_CH4A
IOMUXC_PTB8_PTB8
IOMUXC_PTB8_FXIO0_D24
IOMUXC_PTB8_LPSPI3_PCS2
IOMUXC_PTB8_LPUART3_CTS_B
IOMUXC_PTB8_LPI2C3_SCL
IOMUXC_PTB8_TPM3_CH0
IOMUXC_PTB8_I2S2_TXD1
IOMUXC_PTB8_MQS0_RIGHT

IOMUXC_PTB8_MICFIL0_CLK01
IOMUXC_PTB8_PMIC0_MODE1
IOMUXC_PTB9_ADC0_CH4B
IOMUXC_PTB9_PTB9
IOMUXC_PTB9_FXIO0_D25
IOMUXC_PTB9_LPSPI3_PCS3
IOMUXC_PTB9_LPUART3_RTS_B
IOMUXC_PTB9_LPI2C3_SDA
IOMUXC_PTB9_TPM3_CH1
IOMUXC_PTB9_I2S3_TX_BCLK
IOMUXC_PTB9_MICFIL0_DATA01
IOMUXC_PTB9_PMIC0_MODE0
IOMUXC_PTB10_ADC0_CH5A
IOMUXC_PTB10_PTB10
IOMUXC_PTB10_FXIO0_D26
IOMUXC_PTB10_LPSPI3_SIN
IOMUXC_PTB10_LPUART3_RX
IOMUXC_PTB10_LPI2C3_HREQ
IOMUXC_PTB10_TPM3_CH2
IOMUXC_PTB10_I2S3_TX_FS
IOMUXC_PTB10_MICFIL0_CLK01
IOMUXC_PTB10_PMIC0_SDA
IOMUXC_PTB11_ADC0_CH5B
IOMUXC_PTB11_PTB11
IOMUXC_PTB11_FXIO0_D27
IOMUXC_PTB11_LPSPI3_SOUT
IOMUXC_PTB11_LPUART3_TX
IOMUXC_PTB11_LPI2C3_SDA
IOMUXC_PTB11_TPM3_CH3
IOMUXC_PTB11_I2S3_TXD0
IOMUXC_PTB11_MICFIL0_DATA23
IOMUXC_PTB11_PMIC0_SCL
IOMUXC_PTB12_CMP1_IN0

IOMUXC_PTB12_PTB12
IOMUXC_PTB12_FXIO0_D28
IOMUXC_PTB12_LPSPI3_SCK
IOMUXC_PTB12_LPUART3_CTS_B
IOMUXC_PTB12_LPI2C3_SCL
IOMUXC_PTB12_TPM3_CH4
IOMUXC_PTB12_I2S3_RXD0
IOMUXC_PTB12_MICFIL0_CLK01
IOMUXC_PTB12_WUU0_P24
IOMUXC_PTB13_CMP1_IN1
IOMUXC_PTB13_PTB13
IOMUXC_PTB13_FXIO0_D29
IOMUXC_PTB13_LPSPI3_PCS0
IOMUXC_PTB13_LPUART3_RTS_B
IOMUXC_PTB13_I3C1_SCL
IOMUXC_PTB13_TPM3_CH5
IOMUXC_PTB13_I2S3_RX_BCLK
IOMUXC_PTB13_MICFIL0_DATA45
IOMUXC_PTB13_WDOG2_RST
IOMUXC_PTB13_LPTMR1_ALT3
IOMUXC_PTB13_WUU0_P25
IOMUXC_PTB14_CMP0_IN0
IOMUXC_PTB14_PTB14
IOMUXC_PTB14_FXIO0_D30
IOMUXC_PTB14_LPUART3_TX
IOMUXC_PTB14_I3C1_SDA
IOMUXC_PTB14_I2S3_RX_FS
IOMUXC_PTB14_MICFIL0_CLK01
IOMUXC_PTB14_LPTMR1_ALT2
IOMUXC_PTB14_WUU0_P26
IOMUXC_PTB15_CMP0_IN1
IOMUXC_PTB15_PTB15
IOMUXC_PTB15_FXIO0_D31

IOMUXC_PTB15_LPUART3_RX
IOMUXC_PTB15_I3C1_PUR
IOMUXC_PTB15_I2S3_MCLK
IOMUXC_PTB15_EXT_AUD_MCLK1
IOMUXC_PTB15_MICFIL0_DATA67
IOMUXC_PTB15_LPTMR1_ALT1
IOMUXC_PTB15_WUU0_P27
IOMUXC_PTC0_PTC0
IOMUXC_PTC0_LPSPI2_SIN
IOMUXC_PTC0_FLEXSPI1_B_DQS
IOMUXC_PTC0_TPM2_CLKIN
IOMUXC_PTC0_I2S3_RXD1
IOMUXC_PTC0_FLEXSPI0_A_DQS
IOMUXC_PTC0_MQS0_LEFT
IOMUXC_PTC0_LP_RTD_DBG_MUX_0
IOMUXC_PTC1_PTC1
IOMUXC_PTC1_LPSPI2_SOUT
IOMUXC_PTC1_FLEXSPI1_B_DATA7
IOMUXC_PTC1_TPM2_CH0
IOMUXC_PTC1_I2S3_TXD1
IOMUXC_PTC1_FLEXSPI0_A_DATA7
IOMUXC_PTC1_MQS0_RIGHT
IOMUXC_PTC1_LP_RTD_DBG_MUX_1
IOMUXC_PTC2_PTC2
IOMUXC_PTC2_LPSPI2_SCK
IOMUXC_PTC2_FLEXSPI1_B_DATA6
IOMUXC_PTC2_TPM2_CH1
IOMUXC_PTC2_I2S0_RX_BCLK
IOMUXC_PTC2_FLEXSPI0_A_DATA6
IOMUXC_PTC2_LP_RTD_DBG_MUX_2
IOMUXC_PTC3_PTC3
IOMUXC_PTC3_LPSPI2_PCS0
IOMUXC_PTC3_FLEXSPI1_B_DATA5

IOMUXC_PTC3_I2S0_RX_FS
IOMUXC_PTC3_FLEXSPI0_A_DATA5
IOMUXC_PTC3_LP_RTD_DBG_MUX_3
IOMUXC_PTC4_PTC4
IOMUXC_PTC4_LPSPI2_PCS1
IOMUXC_PTC4_FLEXSPI1_B_DATA4
IOMUXC_PTC4_I2S0_RXD0
IOMUXC_PTC4_FLEXSPI0_A_DATA4
IOMUXC_PTC4_LP_RTD_DBG_MUX_4
IOMUXC_PTC5_PTC5
IOMUXC_PTC5_LPSPI2_PCS2
IOMUXC_PTC5_FLEXSPI1_B_SS0_B
IOMUXC_PTC5_FLEXSPI1_B_SCLK_B
IOMUXC_PTC5_FLEXSPI1_A_SS0_B
IOMUXC_PTC5_I2S0_RXD1
IOMUXC_PTC5_FLEXSPI0_A_SS0_B
IOMUXC_PTC5_FLEXSPI0_A_SCLK_B
IOMUXC_PTC5_LP_RTD_DBG_MUX_5
IOMUXC_PTC6_PTC6
IOMUXC_PTC6_LPSPI2_PCS3
IOMUXC_PTC6_FLEXSPI1_B_SCLK
IOMUXC_PTC6_FLEXSPI1_A_SCLK
IOMUXC_PTC6_I2S0_TXD1
IOMUXC_PTC6_FLEXSPI0_A_SCLK
IOMUXC_PTC6_LP_RTD_DBG_MUX_6
IOMUXC_PTC7_PTC7
IOMUXC_PTC7_FLEXSPI1_B_DATA3
IOMUXC_PTC7_I2S0_TXD0
IOMUXC_PTC7_FLEXSPI0_A_DATA3
IOMUXC_PTC7_LP_RTD_DBG_MUX_7
IOMUXC_PTC8_PTC8
IOMUXC_PTC8_FLEXSPI1_B_DATA2
IOMUXC_PTC8_I2S0_TX_BCLK

IOMUXC_PTC8_FLEXSPI0_A_DATA2
IOMUXC_PTC8_LP_RTD_DBG_MUX_8
IOMUXC_PTC9_PTC9
IOMUXC_PTC9_FLEXSPI1_B_DATA1
IOMUXC_PTC9_I2S0_TX_FS
IOMUXC_PTC9_FLEXSPI0_A_DATA1
IOMUXC_PTC9_LP_RTD_DBG_MUX_9
IOMUXC_PTC10_PTC10
IOMUXC_PTC10_FLEXSPI1_B_DATA0
IOMUXC_PTC10_I2S0_MCLK
IOMUXC_PTC10_FLEXSPI0_A_DATA0
IOMUXC_PTC10_EXT_AUD_MCLK1
IOMUXC_PTC10_LP_RTD_DBG_MUX_10
IOMUXC_PTC11_PTC11
IOMUXC_PTC11_FLEXSPI1_B_SS0_B
IOMUXC_PTC11_FLEXSPI1_B_SS1_B
IOMUXC_PTC11_TPM3_CLKIN
IOMUXC_PTC11_I2S1_RXD3
IOMUXC_PTC11_FLEXSPI0_A_SS0_B
IOMUXC_PTC11_FLEXSPI0_A_SS1_B
IOMUXC_PTC11_CLKOUT0
IOMUXC_PTC11_LP_RTD_DBG_MUX_11
IOMUXC_PTC12_PTC12
IOMUXC_PTC12_FLEXSPI1_A_DQS
IOMUXC_PTC12_TPM3_CH0
IOMUXC_PTC12_I2S1_RXD2
IOMUXC_PTC12_FLEXSPI0_B_DQS
IOMUXC_PTC12_TRACE0_CLKOUT
IOMUXC_PTC12_LP_RTD_DBG_MUX_12
IOMUXC_PTC13_PTC13
IOMUXC_PTC13_LPSPI3_SIN
IOMUXC_PTC13_FLEXSPI1_A_DATA7
IOMUXC_PTC13_TPM3_CH1

IOMUXC_PTC13_I2S1_TXD3
IOMUXC_PTC13_FLEXSPI0_B_DATA7
IOMUXC_PTC13_TRACE0_D0
IOMUXC_PTC13_LP_RTD_DBG_MUX_13
IOMUXC_PTC14_PTC14
IOMUXC_PTC14_LPSPI3_SOUT
IOMUXC_PTC14_FLEXSPI1_A_DATA6
IOMUXC_PTC14_TPM3_CH2
IOMUXC_PTC14_I2S1_TXD2
IOMUXC_PTC14_FLEXSPI0_B_DATA6
IOMUXC_PTC14_TRACE0_D1
IOMUXC_PTC14_LP_RTD_DBG_MUX_14
IOMUXC_PTC15_PTC15
IOMUXC_PTC15_LPSPI3_SCK
IOMUXC_PTC15_FLEXSPI1_A_DATA5
IOMUXC_PTC15_TPM3_CH3
IOMUXC_PTC15_I2S1_RX_BCLK
IOMUXC_PTC15_FLEXSPI0_B_DATA5
IOMUXC_PTC15_TRACE0_D2
IOMUXC_PTC15_LP_RTD_DBG_MUX_15
IOMUXC_PTC16_PTC16
IOMUXC_PTC16_LPSPI3_PCS0
IOMUXC_PTC16_FLEXSPI1_A_DATA4
IOMUXC_PTC16_TPM3_CH4
IOMUXC_PTC16_I2S1_RX_FS
IOMUXC_PTC16_FLEXSPI0_B_DATA4
IOMUXC_PTC16_TRACE0_D3
IOMUXC_PTC16_LP_RTD_DBG_MUX_16
IOMUXC_PTC17_PTC17
IOMUXC_PTC17_LPSPI3_PCS1
IOMUXC_PTC17_FLEXSPI1_A_SS0_B
IOMUXC_PTC17_FLEXSPI1_A_SCLK_B
IOMUXC_PTC17_TPM3_CH5

IOMUXC_PTC17_I2S1_RXD0
IOMUXC_PTC17_FLEXSPI0_B_SS0_B
IOMUXC_PTC17_FLEXSPI0_B_SCLK_B
IOMUXC_PTC17_LP_RTD_DBG_MUX_17
IOMUXC_PTC18_PTC18
IOMUXC_PTC18_LPSPI3_PCS2
IOMUXC_PTC18_FLEXSPI1_A_SCLK
IOMUXC_PTC18_I2S1_RXD1
IOMUXC_PTC18_FLEXSPI0_B_SCLK
IOMUXC_PTC18_LP_RTD_DBG_MUX_18
IOMUXC_PTC19_PTC19
IOMUXC_PTC19_LPSPI3_PCS3
IOMUXC_PTC19_FLEXSPI1_A_DATA3
IOMUXC_PTC19_I2S1_TXD1
IOMUXC_PTC19_FLEXSPI0_B_DATA3
IOMUXC_PTC19_TRACE0_D4
IOMUXC_PTC19_LP_RTD_DBG_MUX_19
IOMUXC_PTC20_PTC20
IOMUXC_PTC20_FLEXSPI1_A_DATA2
IOMUXC_PTC20_I2S1_TXD0
IOMUXC_PTC20_FLEXSPI0_B_DATA2
IOMUXC_PTC20_TRACE0_D5
IOMUXC_PTC21_PTC21
IOMUXC_PTC21_FLEXSPI1_A_DATA1
IOMUXC_PTC21_I2S1_TX_BCLK
IOMUXC_PTC21_FLEXSPI0_B_DATA1
IOMUXC_PTC21_TRACE0_D6
IOMUXC_PTC22_PTC22
IOMUXC_PTC22_FLEXSPI1_A_DATA0
IOMUXC_PTC22_I2S1_TX_FS
IOMUXC_PTC22_FLEXSPI0_B_DATA0
IOMUXC_PTC22_TRACE0_D7
IOMUXC_PTC23_PTC23

IOMUXC_PTC23_FLEXSPI1_A_SS0_B
IOMUXC_PTC23_FLEXSPI1_A_SS1_B
IOMUXC_PTC23_I2S1_MCLK
IOMUXC_PTC23_FLEXSPI0_B_SS0_B
IOMUXC_PTC23_FLEXSPI0_B_SS1_B
IOMUXC_PTC23_CLKOUT0
IOMUXC_PTD0_PTD0
IOMUXC_PTD0_I2S6_RX_BCLK
IOMUXC_PTD0_SDHC0_RESET_B
IOMUXC_PTD0_FLEXSPI2_B_DQS
IOMUXC_PTD0_CLKOUT2
IOMUXC_PTD0_EPDC0_SDCLK_B
IOMUXC_PTD0_LP_APD_DBG_MUX_0
IOMUXC_PTD0_CLKOUT1
IOMUXC_PTD0_DEBUG_MUX0_0
IOMUXC_PTD0_DEBUG_MUX1_0
IOMUXC_PTD1_PTD1
IOMUXC_PTD1_I2S6_RX_FS
IOMUXC_PTD1_SDHC0_CMD
IOMUXC_PTD1_FLEXSPI2_B_DATA7
IOMUXC_PTD1_EPDC0_SDCLK
IOMUXC_PTD1_DPI0_PCLK
IOMUXC_PTD1_LP_APD_DBG_MUX_1
IOMUXC_PTD1_DEBUG_MUX0_1
IOMUXC_PTD1_DEBUG_MUX1_1
IOMUXC_PTD2_PTD2
IOMUXC_PTD2_I2S6_RXD0
IOMUXC_PTD2_SDHC0_CLK
IOMUXC_PTD2_FLEXSPI2_B_DATA6
IOMUXC_PTD2_EPDC0_SDLE
IOMUXC_PTD2_DPI0_HSYNC
IOMUXC_PTD2_LP_APD_DBG_MUX_2
IOMUXC_PTD2_DEBUG_MUX0_2

IOMUXC_PTD2_DEBUG_MUX1_2
IOMUXC_PTD3_PTD3
IOMUXC_PTD3_I2S6_RXD1
IOMUXC_PTD3_SDHC0_D7
IOMUXC_PTD3_FLEXSPI2_B_DATA5
IOMUXC_PTD3_EPDC0_GDSP
IOMUXC_PTD3_DPI0_VSYNC
IOMUXC_PTD3_LP_APD_DBG_MUX_3
IOMUXC_PTD3_DEBUG_MUX0_3
IOMUXC_PTD3_DEBUG_MUX1_3
IOMUXC_PTD4_PTD4
IOMUXC_PTD4_EXT_AUD_MCLK3
IOMUXC_PTD4_SDHC0_VS
IOMUXC_PTD4_TPM8_CH5
IOMUXC_PTD4_I2S6_MCLK
IOMUXC_PTD4_SDHC0_D6
IOMUXC_PTD4_FLEXSPI2_B_DATA4
IOMUXC_PTD4_EPDC0_SDCE0
IOMUXC_PTD4_DPI0_DE
IOMUXC_PTD4_LP_APD_DBG_MUX_4
IOMUXC_PTD4_DEBUG_MUX0_4
IOMUXC_PTD4_DEBUG_MUX1_4
IOMUXC_PTD5_PTD5
IOMUXC_PTD5_SDHC0_CD
IOMUXC_PTD5_TPM8_CH4
IOMUXC_PTD5_I2S6_TX_BCLK
IOMUXC_PTD5_SDHC0_D5
IOMUXC_PTD5_FLEXSPI2_B_SS0_B
IOMUXC_PTD5_FLEXSPI2_B_SCLK_B
IOMUXC_PTD5_EPDC0_D0
IOMUXC_PTD5_DPI0_D0
IOMUXC_PTD5_LP_APD_DBG_MUX_5
IOMUXC_PTD5_DEBUG_MUX0_5

IOMUXC_PTD5_DEBUG_MUX1_5
IOMUXC_PTD6_PTD6
IOMUXC_PTD6_SDHC0_WP
IOMUXC_PTD6_TPM8_CH3
IOMUXC_PTD6_I2S6_TX_FS
IOMUXC_PTD6_SDHC0_D4
IOMUXC_PTD6_FLEXSPI2_B_SCLK
IOMUXC_PTD6_EPDC0_D1
IOMUXC_PTD6_DPI0_D1
IOMUXC_PTD6_LP_APD_DBG_MUX_6
IOMUXC_PTD6_DEBUG_MUX0_6
IOMUXC_PTD6_DEBUG_MUX1_6
IOMUXC_PTD7_PTD7
IOMUXC_PTD7_TPM8_CH2
IOMUXC_PTD7_I2S6_TXD0
IOMUXC_PTD7_SDHC0_D3
IOMUXC_PTD7_FLEXSPI2_B_DATA3
IOMUXC_PTD7_EPDC0_D2
IOMUXC_PTD7_DPI0_D2
IOMUXC_PTD7_LP_APD_DBG_MUX_7
IOMUXC_PTD7_DEBUG_MUX0_7
IOMUXC_PTD7_DEBUG_MUX1_7
IOMUXC_PTD8_PTD8
IOMUXC_PTD8_TPM8_CH1
IOMUXC_PTD8_I2S6_TXD1
IOMUXC_PTD8_SDHC0_D2
IOMUXC_PTD8_FLEXSPI2_B_DATA2
IOMUXC_PTD8_EPDC0_D3
IOMUXC_PTD8_DPI0_D3
IOMUXC_PTD8_LP_APD_DBG_MUX_8
IOMUXC_PTD8_DEBUG_MUX1_8
IOMUXC_PTD9_PTD9
IOMUXC_PTD9_TPM8_CLKIN

IOMUXC_PTD9_I2S6_TXD2
IOMUXC_PTD9_SDHC0_D1
IOMUXC_PTD9_FLEXSPI2_B_DATA1
IOMUXC_PTD9_EPDC0_D4
IOMUXC_PTD9_DPI0_D4
IOMUXC_PTD9_LP_APD_DBG_MUX_9
IOMUXC_PTD9_DEBUG_MUX1_9
IOMUXC_PTD10_PTD10
IOMUXC_PTD10_TPM8_CH0
IOMUXC_PTD10_I2S6_TXD3
IOMUXC_PTD10_SDHC0_D0
IOMUXC_PTD10_FLEXSPI2_B_DATA0
IOMUXC_PTD10_EPDC0_D5
IOMUXC_PTD10_DPI0_D5
IOMUXC_PTD10_LP_APD_DBG_MUX_10
IOMUXC_PTD10_DEBUG_MUX1_10
IOMUXC_PTD11_PTD11
IOMUXC_PTD11_TPM8_CH5
IOMUXC_PTD11_I2S6_RXD2
IOMUXC_PTD11_SDHC0_DQS
IOMUXC_PTD11_FLEXSPI2_B_SS0_B
IOMUXC_PTD11_FLEXSPI2_A_SS1_B
IOMUXC_PTD11_EPDC0_D6
IOMUXC_PTD11_DPI0_D6
IOMUXC_PTD11_LP_APD_DBG_MUX_11
IOMUXC_PTD12_PTD12
IOMUXC_PTD12_USB0_ID
IOMUXC_PTD12_SDHC2_D3
IOMUXC_PTD12_I2S7_RX_BCLK
IOMUXC_PTD12_SDHC1_DQS
IOMUXC_PTD12_FLEXSPI2_A_SS0_B
IOMUXC_PTD12_FLEXSPI2_B_SS1_B
IOMUXC_PTD12_EPDC0_D7

IOMUXC_PTD12_DPI0_D7
IOMUXC_PTD12_LP_APD_DBG_MUX_12
IOMUXC_PTD13_PTD13
IOMUXC_PTD13_SPDIF_IN4
IOMUXC_PTD13_USB0_PWR
IOMUXC_PTD13_SDHC2_D2
IOMUXC_PTD13_I2S7_RX_FS
IOMUXC_PTD13_SDHC1_RESET_B
IOMUXC_PTD13_FLEXSPI2_A_SCLK
IOMUXC_PTD13_CLKOUT2
IOMUXC_PTD13_EPDC0_D8
IOMUXC_PTD13_DPI0_D8
IOMUXC_PTD13_CLKOUT1
IOMUXC_PTD13_LP_APD_DBG_MUX_13
IOMUXC_PTD14_PTD14
IOMUXC_PTD14_SPDIF_SRCLK
IOMUXC_PTD14_USB0_OC
IOMUXC_PTD14_SDHC2_D1
IOMUXC_PTD14_I2S7_RXD0
IOMUXC_PTD14_SDHC1_D7
IOMUXC_PTD14_FLEXSPI2_A_DATA3
IOMUXC_PTD14_TRACE0_D7
IOMUXC_PTD14_EPDC0_D9
IOMUXC_PTD14_DPI0_D9
IOMUXC_PTD14_LP_APD_DBG_MUX_14
IOMUXC_PTD15_PTD15
IOMUXC_PTD15_SPDIF_IN3
IOMUXC_PTD15_SDHC1_VS
IOMUXC_PTD15_SDHC2_D0
IOMUXC_PTD15_I2S7_TX_BCLK
IOMUXC_PTD15_SDHC1_D6
IOMUXC_PTD15_FLEXSPI2_A_DATA2
IOMUXC_PTD15_TRACE0_D6

IOMUXC_PTD15_EPDC0_D10
IOMUXC_PTD15_DPI0_D10
IOMUXC_PTD15_LP_APD_DBG_MUX_15
IOMUXC_PTD16_PTD16
IOMUXC_PTD16_FXIO1_D31
IOMUXC_PTD16_LPSPI4_PCS1
IOMUXC_PTD16_SPDIF_PLOCK
IOMUXC_PTD16_SDHC1_CD
IOMUXC_PTD16_SDHC2_CLK
IOMUXC_PTD16_I2S7_TX_FS
IOMUXC_PTD16_SDHC1_D5
IOMUXC_PTD16_FLEXSPI2_A_DATA1
IOMUXC_PTD16_TRACE0_D5
IOMUXC_PTD16_EPDC0_D11
IOMUXC_PTD16_DPI0_D11
IOMUXC_PTD16_LP_APD_DBG_MUX_16
IOMUXC_PTD17_PTD17
IOMUXC_PTD17_FXIO1_D30
IOMUXC_PTD17_LPSPI4_PCS2
IOMUXC_PTD17_EXT_AUD_MCLK3
IOMUXC_PTD17_SDHC1_WP
IOMUXC_PTD17_SDHC2_CMD
IOMUXC_PTD17_I2S7_TXD0
IOMUXC_PTD17_SDHC1_D4
IOMUXC_PTD17_FLEXSPI2_A_DATA0
IOMUXC_PTD17_TRACE0_D4
IOMUXC_PTD17_EPDC0_D12
IOMUXC_PTD17_DPI0_D12
IOMUXC_PTD17_LP_APD_DBG_MUX_17
IOMUXC_PTD18_PTD18
IOMUXC_PTD18_FXIO1_D29
IOMUXC_PTD18_LPSPI4_PCS3
IOMUXC_PTD18_SPDIF_OUTCLK

IOMUXC_PTD18_EXT_AUD_MCLK3
IOMUXC_PTD18_TPM8_CH0
IOMUXC_PTD18_I2S7_MCLK
IOMUXC_PTD18_SDHC1_D3
IOMUXC_PTD18_FLEXSPI2_A_DQS
IOMUXC_PTD18_TRACE0_D3
IOMUXC_PTD18_EPDC0_D13
IOMUXC_PTD18_DPI0_D13
IOMUXC_PTD18_LP_APD_DBG_MUX_18
IOMUXC_PTD19_PTD19
IOMUXC_PTD19_FXIO1_D28
IOMUXC_PTD19_SPDIF_IN1
IOMUXC_PTD19_TPM8_CH1
IOMUXC_PTD19_I2S6_RXD3
IOMUXC_PTD19_SDHC1_D2
IOMUXC_PTD19_FLEXSPI2_A_DATA7
IOMUXC_PTD19_TRACE0_D2
IOMUXC_PTD19_EPDC0_D14
IOMUXC_PTD19_DPI0_D14
IOMUXC_PTD19_LP_APD_DBG_MUX_19
IOMUXC_PTD20_PTD20
IOMUXC_PTD20_FXIO1_D27
IOMUXC_PTD20_LPSPI4_SIN
IOMUXC_PTD20_SPDIF_OUT1
IOMUXC_PTD20_TPM8_CLKIN
IOMUXC_PTD20_I2S7_RXD1
IOMUXC_PTD20_SDHC1_D1
IOMUXC_PTD20_FLEXSPI2_A_DATA6
IOMUXC_PTD20_TRACE0_D1
IOMUXC_PTD20_EPDC0_D15
IOMUXC_PTD20_DPI0_D15
IOMUXC_PTD20_LP_APD_DBG_MUX_20
IOMUXC_PTD21_PTD21

IOMUXC_PTD21_FXIO1_D26
IOMUXC_PTD21_LPSPI4_SOUT
IOMUXC_PTD21_SPDIF_IN2
IOMUXC_PTD21_USB1_PWR
IOMUXC_PTD21_TPM8_CH2
IOMUXC_PTD21_I2S7_TXD1
IOMUXC_PTD21_SDHC1_D0
IOMUXC_PTD21_FLEXSPI2_A_DATA5
IOMUXC_PTD21_TRACE0_D0
IOMUXC_PTD21_DPI0_D16
IOMUXC_PTD21_WDOG5_RST
IOMUXC_PTD21_LP_APD_DBG_MUX_21
IOMUXC_PTD22_PTD22
IOMUXC_PTD22_FXIO1_D25
IOMUXC_PTD22_LPSPI4_SCK
IOMUXC_PTD22_SPDIF_OUT2
IOMUXC_PTD22_USB1_OC
IOMUXC_PTD22_TPM8_CH3
IOMUXC_PTD22_I2S7_TXD2
IOMUXC_PTD22_SDHC1_CLK
IOMUXC_PTD22_FLEXSPI2_A_DATA4
IOMUXC_PTD22_TRACE0_CLKOUT
IOMUXC_PTD22_DPI0_D17
IOMUXC_PTD22_LP_APD_DBG_MUX_22
IOMUXC_PTD23_PTD23
IOMUXC_PTD23_FXIO1_D24
IOMUXC_PTD23_LPSPI4_PCS0
IOMUXC_PTD23_USB1_ID
IOMUXC_PTD23_TPM8_CH4
IOMUXC_PTD23_I2S7_TXD3
IOMUXC_PTD23_SDHC1_CMD
IOMUXC_PTD23_FLEXSPI2_A_SS0_B
IOMUXC_PTD23_FLEXSPI2_A_SCLK_B

IOMUXC_PTD23_DPI0_D18
IOMUXC_PTD23_LP_APD_DBG_MUX_23
IOMUXC_PTE0_PTE0
IOMUXC_PTE0_FXIO1_D23
IOMUXC_PTE0_SPDIF_IN4
IOMUXC_PTE0_LPUART4_CTS_B
IOMUXC_PTE0_LPI2C4_SCL
IOMUXC_PTE0_TPM8_CLKIN
IOMUXC_PTE0_I2S7_RXD2
IOMUXC_PTE0_SDHC2_D1
IOMUXC_PTE0_FLEXSPI2_B_DQS
IOMUXC_PTE0_ENET0_CRS
IOMUXC_PTE0_DBI0_WRX
IOMUXC_PTE0_DPI0_D19
IOMUXC_PTE0_WUU1_P0
IOMUXC_PTE0_DEBUG_MUX0_8
IOMUXC_PTE0_DEBUG_MUX1_11
IOMUXC_PTE1_PTE1
IOMUXC_PTE1_FXIO1_D22
IOMUXC_PTE1_SPDIF_SRCLK
IOMUXC_PTE1_LPUART4_RTS_B
IOMUXC_PTE1_LPI2C4_SDA
IOMUXC_PTE1_TPM8_CH0
IOMUXC_PTE1_I2S7_RXD3
IOMUXC_PTE1_SDHC2_D0
IOMUXC_PTE1_FLEXSPI2_B_DATA7
IOMUXC_PTE1_ENET0_COL
IOMUXC_PTE1_DBI0_CSX
IOMUXC_PTE1_DPI0_D20
IOMUXC_PTE1_WUU1_P1
IOMUXC_PTE1_DEBUG_MUX0_9
IOMUXC_PTE1_DEBUG_MUX1_12
IOMUXC_PTE2_PTE2

IOMUXC_PTE2_FXIO1_D21
IOMUXC_PTE2_SPDIF_IN3
IOMUXC_PTE2_LPUART4_TX
IOMUXC_PTE2_LPI2C4_HREQ
IOMUXC_PTE2_TPM8_CH1
IOMUXC_PTE2_EXT_AUD_MCLK3
IOMUXC_PTE2_SDHC2_CLK
IOMUXC_PTE2_FLEXSPI2_B_DATA6
IOMUXC_PTE2_ENET0_TXER
IOMUXC_PTE2_DBI0_DCX
IOMUXC_PTE2_DPI0_D21
IOMUXC_PTE2_LP_HV_DBG_MUX_0
IOMUXC_PTE2_DEBUG_MUX0_10
IOMUXC_PTE2_DEBUG_MUX1_13
IOMUXC_PTE3_PTE3
IOMUXC_PTE3_FXIO1_D20
IOMUXC_PTE3_SPDIF_PLOCK
IOMUXC_PTE3_LPUART4_RX
IOMUXC_PTE3_TPM8_CH2
IOMUXC_PTE3_I2S6_MCLK
IOMUXC_PTE3_SDHC2_CMD
IOMUXC_PTE3_FLEXSPI2_B_DATA5
IOMUXC_PTE3_ENET0_TXCLK
IOMUXC_PTE3_DBI0_RWX
IOMUXC_PTE3_DPI0_D22
IOMUXC_PTE3_WUU1_P2
IOMUXC_PTE3_DEBUG_MUX0_11
IOMUXC_PTE3_DEBUG_MUX1_14
IOMUXC_PTE4_PTE4
IOMUXC_PTE4_FXIO1_D19
IOMUXC_PTE4_SPDIF_OUTCLK
IOMUXC_PTE4_LPUART5_CTS_B
IOMUXC_PTE4_LPI2C5_SCL

IOMUXC_PTE4_TPM8_CH3
IOMUXC_PTE4_I2S6_RX_BCLK
IOMUXC_PTE4_SDHC2_D3
IOMUXC_PTE4_FLEXSPI2_B_DATA4
IOMUXC_PTE4_ENET0_TXD3
IOMUXC_PTE4_DBI0_E
IOMUXC_PTE4_DPI0_D23
IOMUXC_PTE4_WUU1_P3
IOMUXC_PTE4_DEBUG_MUX0_12
IOMUXC_PTE4_DEBUG_MUX1_15
IOMUXC_PTE5_PTE5
IOMUXC_PTE5_FXIO1_D18
IOMUXC_PTE5_SPDIF_IN1
IOMUXC_PTE5_LPUART5_RTS_B
IOMUXC_PTE5_LPI2C5_SDA
IOMUXC_PTE5_TPM8_CH4
IOMUXC_PTE5_I2S6_RX_FS
IOMUXC_PTE5_SDHC2_D2
IOMUXC_PTE5_FLEXSPI2_B_SS0_B
IOMUXC_PTE5_ENET0_TXD2
IOMUXC_PTE5_DBI0_D0
IOMUXC_PTE5_LP_HV_DBG_MUX_1
IOMUXC_PTE5_DEBUG_MUX0_13
IOMUXC_PTE5_DEBUG_MUX1_16
IOMUXC_PTE6_PTE6
IOMUXC_PTE6_FXIO1_D17
IOMUXC_PTE6_SPDIF_OUT1
IOMUXC_PTE6_LPUART5_TX
IOMUXC_PTE6_LPI2C5_HREQ
IOMUXC_PTE6_TPM8_CH5
IOMUXC_PTE6_I2S6_RXD0
IOMUXC_PTE6_SDHC2_D4
IOMUXC_PTE6_FLEXSPI2_B_SCLK

IOMUXC_PTE6_ENET0_RXCLK
IOMUXC_PTE6_DBI0_D1
IOMUXC_PTE6_LP_HV_DBG_MUX_2
IOMUXC_PTE6_WDOG5_RST
IOMUXC_PTE6_DEBUG_MUX0_14
IOMUXC_PTE6_DEBUG_MUX1_17
IOMUXC_PTE7_PTE7
IOMUXC_PTE7_FXIO1_D16
IOMUXC_PTE7_SPDIF_IN2
IOMUXC_PTE7_LPUART5_RX
IOMUXC_PTE7_LPI2C6_HREQ
IOMUXC_PTE7_TPM4_CLKIN
IOMUXC_PTE7_I2S6_RXD1
IOMUXC_PTE7_SDHC2_D5
IOMUXC_PTE7_FLEXSPI2_B_DATA3
IOMUXC_PTE7_ENET0_RXD3
IOMUXC_PTE7_DBI0_D2
IOMUXC_PTE7_EPDC0_BDR1
IOMUXC_PTE7_WUU1_P4
IOMUXC_PTE7_DEBUG_MUX0_15
IOMUXC_PTE7_DEBUG_MUX1_18
IOMUXC_PTE8_PTE8
IOMUXC_PTE8_FXIO1_D15
IOMUXC_PTE8_LPSPI4_PCS1
IOMUXC_PTE8_LPUART6_CTS_B
IOMUXC_PTE8_LPI2C6_SCL
IOMUXC_PTE8_TPM4_CH0
IOMUXC_PTE8_I2S6_RXD2
IOMUXC_PTE8_SDHC2_D6
IOMUXC_PTE8_FLEXSPI2_B_DATA2
IOMUXC_PTE8_ENET0_RXD2
IOMUXC_PTE8_DBI0_D3
IOMUXC_PTE8_EPDC0_BDR0

IOMUXC_PTE8_LP_HV_DBG_MUX_3
IOMUXC_PTE8_DEBUG_MUX1_19
IOMUXC_PTE9_PTE9
IOMUXC_PTE9_FXIO1_D14
IOMUXC_PTE9_LPSPI4_PCS2
IOMUXC_PTE9_LPUART6_RTS_B
IOMUXC_PTE9_LPI2C6_SDA
IOMUXC_PTE9_TPM4_CH1
IOMUXC_PTE9_I2S6_RXD3
IOMUXC_PTE9_SDHC2_D7
IOMUXC_PTE9_FLEXSPI2_B_DATA1
IOMUXC_PTE9_ENET0_1588_TMR3
IOMUXC_PTE9_DBI0_D4
IOMUXC_PTE9_EPDC0_VCOM1
IOMUXC_PTE9_LP_HV_DBG_MUX_4
IOMUXC_PTE9_DEBUG_MUX1_20
IOMUXC_PTE10_PTE10
IOMUXC_PTE10_FXIO1_D13
IOMUXC_PTE10_LPSPI4_PCS3
IOMUXC_PTE10_LPUART6_TX
IOMUXC_PTE10_I3C2_SCL
IOMUXC_PTE10_TPM4_CH2
IOMUXC_PTE10_I2S6_TX_BCLK
IOMUXC_PTE10_SDHC2_DQS
IOMUXC_PTE10_FLEXSPI2_B_DATA0
IOMUXC_PTE10_ENET0_1588_TMR2
IOMUXC_PTE10_DBI0_D5
IOMUXC_PTE10_EPDC0_VCOM0
IOMUXC_PTE10_LP_HV_DBG_MUX_5
IOMUXC_PTE10_DEBUG_MUX1_21
IOMUXC_PTE11_PTE11
IOMUXC_PTE11_FXIO1_D12
IOMUXC_PTE11_SPDIF_OUT2

IOMUXC_PTE11_LPUART6_RX
IOMUXC_PTE11_I3C2_SDA
IOMUXC_PTE11_TPM4_CH3
IOMUXC_PTE11_I2S6_TX_FS
IOMUXC_PTE11_FLEXSPI2_B_SCLK_B
IOMUXC_PTE11_FLEXSPI2_B_SS0_B
IOMUXC_PTE11_ENET0_1588_TMR1
IOMUXC_PTE11_DBI0_D6
IOMUXC_PTE11_EPDC0_PWRCTRL0
IOMUXC_PTE11_LP_HV_DBG_MUX_6
IOMUXC_PTE12_PTE12
IOMUXC_PTE12_FXIO1_D11
IOMUXC_PTE12_LPSPI4_SIN
IOMUXC_PTE12_LPUART7_CTS_B
IOMUXC_PTE12_LPI2C7_SCL
IOMUXC_PTE12_TPM4_CH4
IOMUXC_PTE12_I2S6_TXD0
IOMUXC_PTE12_SDHC2_RESET_B
IOMUXC_PTE12_FLEXSPI2_B_SS1_B
IOMUXC_PTE12_ENET0_1588_TMR0
IOMUXC_PTE12_DBI0_D7
IOMUXC_PTE12_EPDC0_PWRCTRL1
IOMUXC_PTE12_WUU1_P5
IOMUXC_PTE13_PTE13
IOMUXC_PTE13_FXIO1_D10
IOMUXC_PTE13_LPSPI4_SOUT
IOMUXC_PTE13_LPUART7_RTS_B
IOMUXC_PTE13_LPI2C7_SDA
IOMUXC_PTE13_TPM4_CH5
IOMUXC_PTE13_I2S6_TXD1
IOMUXC_PTE13_SDHC1_WP
IOMUXC_PTE13_ENET0_1588_CLKIN
IOMUXC_PTE13_DBI0_D8

IOMUXC_PTE13_EPDC0_PWRCTRL2
IOMUXC_PTE13_LP_HV_DBG_MUX_7
IOMUXC_PTE14_PTE14
IOMUXC_PTE14_FXIO1_D9
IOMUXC_PTE14_LPSPI4_SCK
IOMUXC_PTE14_LPUART7_TX
IOMUXC_PTE14_LPI2C7_HREQ
IOMUXC_PTE14_TPM5_CLKIN
IOMUXC_PTE14_I2S6_TXD2
IOMUXC_PTE14_SDHC1_CD
IOMUXC_PTE14_ENET0_MDIO
IOMUXC_PTE14_DBI0_D9
IOMUXC_PTE14_EPDC0_PWRCTRL3
IOMUXC_PTE14_LP_HV_DBG_MUX_8
IOMUXC_PTE15_PTE15
IOMUXC_PTE15_FXIO1_D8
IOMUXC_PTE15_LPSPI4_PCS0
IOMUXC_PTE15_LPUART7_RX
IOMUXC_PTE15_I3C2_PUR
IOMUXC_PTE15_TPM5_CH0
IOMUXC_PTE15_I2S6_TXD3
IOMUXC_PTE15_MQS1_LEFT
IOMUXC_PTE15_ENET0_MDC
IOMUXC_PTE15_DBI0_D10
IOMUXC_PTE15_EPDC0_PWRCOM
IOMUXC_PTE15_WUU1_P6
IOMUXC_PTE16_PTE16
IOMUXC_PTE16_FXIO1_D7
IOMUXC_PTE16_LPSPI5_PCS1
IOMUXC_PTE16_LPUART4_CTS_B
IOMUXC_PTE16_LPI2C4_SCL
IOMUXC_PTE16_TPM5_CH1
IOMUXC_PTE16_MQS1_LEFT

IOMUXC_PTE16_MQS1_RIGHT
IOMUXC_PTE16_USB0_ID
IOMUXC_PTE16_ENET0_TXEN
IOMUXC_PTE16_DBI0_D11
IOMUXC_PTE16_EPDC0_PWRIRQ
IOMUXC_PTE16_WDOG3_RST
IOMUXC_PTE16_LP_HV_DBG_MUX_9
IOMUXC_PTE17_PTE17
IOMUXC_PTE17_FXIO1_D6
IOMUXC_PTE17_LPSPI5_PCS2
IOMUXC_PTE17_LPUART4_RTS_B
IOMUXC_PTE17_LPI2C4_SDA
IOMUXC_PTE17_MQS1_RIGHT
IOMUXC_PTE17_SDHC1_VS
IOMUXC_PTE17_USB0_PWR
IOMUXC_PTE17_ENET0_RXER
IOMUXC_PTE17_DBI0_D12
IOMUXC_PTE17_EPDC0_PWRSTAT
IOMUXC_PTE17_LP_HV_DBG_MUX_10
IOMUXC_PTE18_PTE18
IOMUXC_PTE18_FXIO1_D5
IOMUXC_PTE18_LPSPI5_PCS3
IOMUXC_PTE18_LPUART4_TX
IOMUXC_PTE18_LPI2C4_HREQ
IOMUXC_PTE18_I2S7_TX_BCLK
IOMUXC_PTE18_USB0_OC
IOMUXC_PTE18_ENET0_CRS_DV
IOMUXC_PTE18_DBI0_D13
IOMUXC_PTE18_EPDC0_PWRWAKE
IOMUXC_PTE18_LP_HV_DBG_MUX_11
IOMUXC_PTE19_PTE19
IOMUXC_PTE19_FXIO1_D4
IOMUXC_PTE19_LPUART4_RX

IOMUXC_PTE19_LPI2C5_HREQ
IOMUXC_PTE19_I3C2_PUR
IOMUXC_PTE19_I2S7_TX_FS
IOMUXC_PTE19_USB1_PWR
IOMUXC_PTE19_ENET0_REFCLK
IOMUXC_PTE19_DBI0_D14
IOMUXC_PTE19_EPDC0_GDCLK
IOMUXC_PTE19_WUU1_P7
IOMUXC_PTE20_PTE20
IOMUXC_PTE20_FXIO1_D3
IOMUXC_PTE20_LPSPI5_SIN
IOMUXC_PTE20_LPUART5_CTS_B
IOMUXC_PTE20_LPI2C5_SCL
IOMUXC_PTE20_I2S7_TXD0
IOMUXC_PTE20_USB1_OC
IOMUXC_PTE20_ENET0_RXD1
IOMUXC_PTE20_DBI0_D15
IOMUXC_PTE20_EPDC0_GDOE
IOMUXC_PTE20_LP_HV_DBG_MUX_12
IOMUXC_PTE21_PTE21
IOMUXC_PTE21_FXIO1_D2
IOMUXC_PTE21_LPSPI5_SOUT
IOMUXC_PTE21_LPUART5_RTS_B
IOMUXC_PTE21_LPI2C5_SDA
IOMUXC_PTE21_TPM6_CLKIN
IOMUXC_PTE21_I2S7_TXD1
IOMUXC_PTE21_USB1_ID
IOMUXC_PTE21_ENET0_RXD0
IOMUXC_PTE21_EPDC0_GDRL
IOMUXC_PTE21_WDOG4_RST
IOMUXC_PTE21_LP_HV_DBG_MUX_13
IOMUXC_PTE22_PTE22
IOMUXC_PTE22_FXIO1_D1

IOMUXC_PTE22_LPSPi5_SCK
IOMUXC_PTE22_LPUART5_TX
IOMUXC_PTE22_I3C2_SCL
IOMUXC_PTE22_TPM6_CH0
IOMUXC_PTE22_I2S7_TXD2
IOMUXC_PTE22_EXT_AUD_MCLK3
IOMUXC_PTE22_ENET0_TXD1
IOMUXC_PTE22_EPDC0_SDOED
IOMUXC_PTE22_CLKOUT2
IOMUXC_PTE22_LP_HV_DBG_MUX_14
IOMUXC_PTE23_PTE23
IOMUXC_PTE23_FXIO1_D0
IOMUXC_PTE23_LPSPi5_PCS0
IOMUXC_PTE23_LPUART5_RX
IOMUXC_PTE23_I3C2_SDA
IOMUXC_PTE23_TPM6_CH1
IOMUXC_PTE23_I2S7_TXD3
IOMUXC_PTE23_EXT_AUD_MCLK2
IOMUXC_PTE23_ENET0_TXD0
IOMUXC_PTE23_EPDC0_SDOEZ
IOMUXC_PTE23_CLKOUT1
IOMUXC_PTE23_LP_HV_DBG_MUX_15
IOMUXC_PTF0_PTF0
IOMUXC_PTF0_FXIO1_D0
IOMUXC_PTF0_LPUART6_CTS_B
IOMUXC_PTF0_LPI2C6_SCL
IOMUXC_PTF0_I2S7_RX_BCLK
IOMUXC_PTF0_SDHC1_D1
IOMUXC_PTF0_ENET0_RXD1
IOMUXC_PTF0_USB1_ID
IOMUXC_PTF0_EPDC0_SDOE
IOMUXC_PTF0_DPI0_D23
IOMUXC_PTF0_WUU1_P8

IOMUXC_PTF1_PTF1
IOMUXC_PTF1_FXIO1_D1
IOMUXC_PTF1_LPUART6_RTS_B
IOMUXC_PTF1_LPI2C6_SDA
IOMUXC_PTF1_I2S7_RX_FS
IOMUXC_PTF1_SDHC1_D0
IOMUXC_PTF1_ENET0_RXD0
IOMUXC_PTF1_LP_HV_DBG_MUX_16
IOMUXC_PTF1_EPDC0_SDSHR
IOMUXC_PTF1_DPI0_D22
IOMUXC_PTF1_WDOG3_RST
IOMUXC_PTF1_DEBUG_MUX0_16
IOMUXC_PTF1_DEBUG_MUX1_22
IOMUXC_PTF2_PTF2
IOMUXC_PTF2_FXIO1_D2
IOMUXC_PTF2_LPUART6_TX
IOMUXC_PTF2_LPI2C6_HREQ
IOMUXC_PTF2_I2S7_RXD0
IOMUXC_PTF2_SDHC1_CLK
IOMUXC_PTF2_ENET0_TXD1
IOMUXC_PTF2_USB0_ID
IOMUXC_PTF2_EPDC0_SDCE9
IOMUXC_PTF2_DPI0_D21
IOMUXC_PTF2_LP_HV_DBG_MUX_17
IOMUXC_PTF2_DEBUG_MUX0_17
IOMUXC_PTF2_DEBUG_MUX1_23
IOMUXC_PTF3_PTF3
IOMUXC_PTF3_FXIO1_D3
IOMUXC_PTF3_LPUART6_RX
IOMUXC_PTF3_LPI2C7_HREQ
IOMUXC_PTF3_I2S7_RXD1
IOMUXC_PTF3_SDHC1_CMD
IOMUXC_PTF3_ENET0_TXD0

IOMUXC_PTF3_USB0_PWR
IOMUXC_PTF3_EPDC0_SDCE8
IOMUXC_PTF3_DPI0_D20
IOMUXC_PTF3_WUU1_P9
IOMUXC_PTF3_DEBUG_MUX1_24
IOMUXC_PTF4_PTF4
IOMUXC_PTF4_FXIO1_D4
IOMUXC_PTF4_LPSPI4_PCS1
IOMUXC_PTF4_LPUART7_CTS_B
IOMUXC_PTF4_LPI2C7_SCL
IOMUXC_PTF4_TPM7_CLKIN
IOMUXC_PTF4_I2S7_RXD2
IOMUXC_PTF4_SDHC1_D3
IOMUXC_PTF4_ENET0_TXEN
IOMUXC_PTF4_USB0_OC
IOMUXC_PTF4_EPDC0_SDCE7
IOMUXC_PTF4_DPI0_D19
IOMUXC_PTF4_WUU1_P10
IOMUXC_PTF4_DEBUG_MUX1_25
IOMUXC_PTF5_PTF5
IOMUXC_PTF5_FXIO1_D5
IOMUXC_PTF5_LPSPI4_PCS2
IOMUXC_PTF5_LPUART7_RTS_B
IOMUXC_PTF5_LPI2C7_SDA
IOMUXC_PTF5_TPM7_CH0
IOMUXC_PTF5_I2S7_RXD3
IOMUXC_PTF5_SDHC1_D2
IOMUXC_PTF5_ENET0_RXER
IOMUXC_PTF5_USB1_PWR
IOMUXC_PTF5_EPDC0_SDCE6
IOMUXC_PTF5_DPI0_D18
IOMUXC_PTF5_LP_HV_DBG_MUX_18
IOMUXC_PTF5_DEBUG_MUX0_18

IOMUXC_PTF5_DEBUG_MUX1_26
IOMUXC_PTF6_LP_HV_DBG_MUX_19
IOMUXC_PTF6_PTF6
IOMUXC_PTF6_FXIO1_D6
IOMUXC_PTF6_LPSP14_PCS3
IOMUXC_PTF6_LPUART7_TX
IOMUXC_PTF6_I3C2_SCL
IOMUXC_PTF6_TPM7_CH1
IOMUXC_PTF6_I2S7_MCLK
IOMUXC_PTF6_SDHC1_D4
IOMUXC_PTF6_ENET0_CRS_DV
IOMUXC_PTF6_USB1_OC
IOMUXC_PTF6_EPDC0_SDCE5
IOMUXC_PTF6_DPI0_D17
IOMUXC_PTF6_WDOG4_RST
IOMUXC_PTF6_DEBUG_MUX0_19
IOMUXC_PTF6_DEBUG_MUX1_27
IOMUXC_PTF7_PTF7
IOMUXC_PTF7_FXIO1_D7
IOMUXC_PTF7_LPUART7_RX
IOMUXC_PTF7_I3C2_SDA
IOMUXC_PTF7_TPM7_CH2
IOMUXC_PTF7_MQS1_LEFT
IOMUXC_PTF7_SDHC1_D5
IOMUXC_PTF7_ENET0_REFCLK
IOMUXC_PTF7_TRACE0_D15
IOMUXC_PTF7_EPDC0_SDCE4
IOMUXC_PTF7_DPI0_D16
IOMUXC_PTF7_WUU1_P11
IOMUXC_PTF7_DEBUG_MUX1_28
IOMUXC_PTF8_PTF8
IOMUXC_PTF8_FXIO1_D8
IOMUXC_PTF8_LPSP14_SIN

IOMUXC_PTF8_LPUART4_CTS_B
IOMUXC_PTF8_LPI2C4_SCL
IOMUXC_PTF8_TPM7_CH3
IOMUXC_PTF8_MQS1_RIGHT
IOMUXC_PTF8_SDHC1_D6
IOMUXC_PTF8_ENET0_MDIO
IOMUXC_PTF8_TRACE0_D14
IOMUXC_PTF8_EPDC0_D15
IOMUXC_PTF8_DPI0_D15
IOMUXC_PTF8_LP_HV_DBG_MUX_24
IOMUXC_PTF8_DEBUG_MUX1_29
IOMUXC_PTF9_PTF9
IOMUXC_PTF9_FXIO1_D9
IOMUXC_PTF9_LPSPI4_SOUT
IOMUXC_PTF9_LPUART4_RTS_B
IOMUXC_PTF9_LPI2C4_SDA
IOMUXC_PTF9_TPM7_CH4
IOMUXC_PTF9_EXT_AUD_MCLK2
IOMUXC_PTF9_SDHC1_D7
IOMUXC_PTF9_ENET0_MDC
IOMUXC_PTF9_TRACE0_D13
IOMUXC_PTF9_EPDC0_D14
IOMUXC_PTF9_DPI0_D14
IOMUXC_PTF9_LP_HV_DBG_MUX_25
IOMUXC_PTF9_DEBUG_MUX1_30
IOMUXC_PTF10_LP_HV_DBG_MUX_26
IOMUXC_PTF10_PTF10
IOMUXC_PTF10_FXIO1_D10
IOMUXC_PTF10_LPSPI4_SCK
IOMUXC_PTF10_LPUART4_TX
IOMUXC_PTF10_LPI2C4_HREQ
IOMUXC_PTF10_TPM7_CH5
IOMUXC_PTF10_I2S4_RX_BCLK

IOMUXC_PTF10_SDHC1_DQS
IOMUXC_PTF10_ENET0_1588_CLKIN
IOMUXC_PTF10_TRACE0_D12
IOMUXC_PTF10_EPDC0_D13
IOMUXC_PTF10_DPI0_D13
IOMUXC_PTF10_DEBUG_MUX0_20
IOMUXC_PTF10_DEBUG_MUX1_31
IOMUXC_PTF11_PTF11
IOMUXC_PTF11_FXIO1_D11
IOMUXC_PTF11_LPSPI4_PCS0
IOMUXC_PTF11_LPUART4_RX
IOMUXC_PTF11_TPM4_CLKIN
IOMUXC_PTF11_I2S4_RX_FS
IOMUXC_PTF11_SDHC1_RESET_B
IOMUXC_PTF11_ENET0_1588_TMR0
IOMUXC_PTF11_TRACE0_D11
IOMUXC_PTF11_EPDC0_D12
IOMUXC_PTF11_DPI0_D12
IOMUXC_PTF11_LP_HV_DBG_MUX_27
IOMUXC_PTF11_DEBUG_MUX1_32
IOMUXC_PTF12_PTF12
IOMUXC_PTF12_FXIO1_D12
IOMUXC_PTF12_LPSPI5_PCS1
IOMUXC_PTF12_LPUART5_CTS_B
IOMUXC_PTF12_LPI2C5_SCL
IOMUXC_PTF12_TPM4_CH0
IOMUXC_PTF12_I2S4_RXD0
IOMUXC_PTF12_SDHC2_WP
IOMUXC_PTF12_ENET0_1588_TMR1
IOMUXC_PTF12_TRACE0_D10
IOMUXC_PTF12_EPDC0_D11
IOMUXC_PTF12_DPI0_D11
IOMUXC_PTF12_LP_HV_DBG_MUX_28

IOMUXC_PTF12_DEBUG_MUX1_33
IOMUXC_PTF13_PTF13
IOMUXC_PTF13_FXIO1_D13
IOMUXC_PTF13_LPSPI5_PCS2
IOMUXC_PTF13_LPUART5_RTS_B
IOMUXC_PTF13_LPI2C5_SDA
IOMUXC_PTF13_TPM4_CH1
IOMUXC_PTF13_I2S4_RXD1
IOMUXC_PTF13_SDHC2_CD
IOMUXC_PTF13_ENET0_1588_TMR2
IOMUXC_PTF13_TRACE0_D9
IOMUXC_PTF13_EPDC0_D10
IOMUXC_PTF13_DPI0_D10
IOMUXC_PTF13_DEBUG_MUX0_21
IOMUXC_PTF13_LP_HV_DBG_MUX_29
IOMUXC_PTF14_PTF14
IOMUXC_PTF14_FXIO1_D14
IOMUXC_PTF14_LPSPI5_PCS3
IOMUXC_PTF14_LPUART5_TX
IOMUXC_PTF14_LPI2C5_HREQ
IOMUXC_PTF14_TPM4_CH2
IOMUXC_PTF14_I2S4_MCLK
IOMUXC_PTF14_SDHC2_VS
IOMUXC_PTF14_ENET0_1588_TMR3
IOMUXC_PTF14_TRACE0_D8
IOMUXC_PTF14_EPDC0_D9
IOMUXC_PTF14_DPI0_D9
IOMUXC_PTF14_DEBUG_MUX0_22
IOMUXC_PTF14_LP_HV_DBG_MUX_30
IOMUXC_PTF15_PTF15
IOMUXC_PTF15_FXIO1_D15
IOMUXC_PTF15_LPUART5_RX
IOMUXC_PTF15_TPM4_CH3

IOMUXC_PTF15_I2S4_TX_BCLK
IOMUXC_PTF15_SDHC2_D1
IOMUXC_PTF15_ENET0_RXD2
IOMUXC_PTF15_TRACE0_D7
IOMUXC_PTF15_EPDC0_D8
IOMUXC_PTF15_DPI0_D8
IOMUXC_PTF15_LP_HV_DBG_MUX_31
IOMUXC_PTF16_PTF16
IOMUXC_PTF16_FXIO1_D16
IOMUXC_PTF16_LPSPI5_SIN
IOMUXC_PTF16_LPUART6_CTS_B
IOMUXC_PTF16_LPI2C6_SCL
IOMUXC_PTF16_TPM4_CH4
IOMUXC_PTF16_I2S4_TX_FS
IOMUXC_PTF16_SDHC2_D0
IOMUXC_PTF16_ENET0_RXD3
IOMUXC_PTF16_TRACE0_D6
IOMUXC_PTF16_EPDC0_D7
IOMUXC_PTF16_DPI0_D7
IOMUXC_PTF16_LP_HV_DBG_MUX_32
IOMUXC_PTF17_PTF17
IOMUXC_PTF17_FXIO1_D17
IOMUXC_PTF17_LPSPI5_SOUT
IOMUXC_PTF17_LPUART6_RTS_B
IOMUXC_PTF17_LPI2C6_SDA
IOMUXC_PTF17_TPM4_CH5
IOMUXC_PTF17_I2S4_TXD0
IOMUXC_PTF17_SDHC2_CLK
IOMUXC_PTF17_ENET0_RXCLK
IOMUXC_PTF17_TRACE0_D5
IOMUXC_PTF17_EPDC0_D6
IOMUXC_PTF17_DPI0_D6
IOMUXC_PTF17_DEBUG_MUX0_23

IOMUXC_PTF17_LP_HV_DBG_MUX_33

IOMUXC_PTF18_PTF18

IOMUXC_PTF18_FXIO1_D18

IOMUXC_PTF18_LPSPI5_SCK

IOMUXC_PTF18_LPUART6_TX

IOMUXC_PTF18_LPI2C6_HREQ

IOMUXC_PTF18_TPM5_CLKIN

IOMUXC_PTF18_I2S4_TXD1

IOMUXC_PTF18_SDHC2_CMD

IOMUXC_PTF18_ENET0_TXD2

IOMUXC_PTF18_TRACE0_D4

IOMUXC_PTF18_EPDC0_D5

IOMUXC_PTF18_DPI0_D5

IOMUXC_PTF19_PTF19

IOMUXC_PTF19_FXIO1_D19

IOMUXC_PTF19_LPSPI5_PCS0

IOMUXC_PTF19_LPUART6_RX

IOMUXC_PTF19_TPM5_CH0

IOMUXC_PTF19_I2S5_RX_BCLK

IOMUXC_PTF19_SDHC2_D3

IOMUXC_PTF19_ENET0_TXD3

IOMUXC_PTF19_TRACE0_D3

IOMUXC_PTF19_EPDC0_D4

IOMUXC_PTF19_DPI0_D4

IOMUXC_PTF20_PTF20

IOMUXC_PTF20_FXIO1_D20

IOMUXC_PTF20_LPUART7_CTS_B

IOMUXC_PTF20_LPI2C7_SCL

IOMUXC_PTF20_TPM5_CH1

IOMUXC_PTF20_I2S5_RX_FS

IOMUXC_PTF20_SDHC2_D2

IOMUXC_PTF20_ENET0_TXCLK

IOMUXC_PTF20_TRACE0_D2

IOMUXC_PTF20_EPDC0_D3
IOMUXC_PTF20_DPI0_D3
IOMUXC_PTF21_PTF21
IOMUXC_PTF21_FXIO1_D21
IOMUXC_PTF21_SPDIF_OUTCLK
IOMUXC_PTF21_LPUART7_RTS_B
IOMUXC_PTF21_LPI2C7_SDA
IOMUXC_PTF21_TPM6_CLKIN
IOMUXC_PTF21_I2S5_RXD0
IOMUXC_PTF21_SDHC2_D4
IOMUXC_PTF21_ENET0_CRS
IOMUXC_PTF21_TRACE0_D1
IOMUXC_PTF21_EPDC0_D2
IOMUXC_PTF21_DPI0_D2
IOMUXC_PTF22_PTF22
IOMUXC_PTF22_FXIO1_D22
IOMUXC_PTF22_SPDIF_IN1
IOMUXC_PTF22_LPUART7_TX
IOMUXC_PTF22_LPI2C7_HREQ
IOMUXC_PTF22_TPM6_CH0
IOMUXC_PTF22_I2S5_RXD1
IOMUXC_PTF22_SDHC2_D5
IOMUXC_PTF22_ENET0_COL
IOMUXC_PTF22_TRACE0_D0
IOMUXC_PTF22_EPDC0_D1
IOMUXC_PTF22_DPI0_D1
IOMUXC_PTF23_PTF23
IOMUXC_PTF23_FXIO1_D23
IOMUXC_PTF23_SPDIF_OUT1
IOMUXC_PTF23_LPUART7_RX
IOMUXC_PTF23_I3C2_PUR
IOMUXC_PTF23_TPM6_CH1
IOMUXC_PTF23_I2S5_RXD2

IOMUXC_PTF23_SDHC2_D6
IOMUXC_PTF23_ENET0_TXER
IOMUXC_PTF23_TRACE0_CLKOUT
IOMUXC_PTF23_EPDC0_D0
IOMUXC_PTF23_DPI0_D0
IOMUXC_PTF24_PTF24
IOMUXC_PTF24_FXIO1_D24
IOMUXC_PTF24_SPDIF_IN2
IOMUXC_PTF24_I3C2_SCL
IOMUXC_PTF24_I2S5_RXD3
IOMUXC_PTF24_SDHC2_D7
IOMUXC_PTF24_DBI0_WRX
IOMUXC_PTF24_EPDC0_SDCLK
IOMUXC_PTF24_DPI0_PCLK
IOMUXC_PTF24_WUU1_P12
IOMUXC_PTF25_PTF25
IOMUXC_PTF25_FXIO1_D25
IOMUXC_PTF25_SPDIF_OUT2
IOMUXC_PTF25_I3C2_SDA
IOMUXC_PTF25_TPM7_CH5
IOMUXC_PTF25_I2S5_MCLK
IOMUXC_PTF25_SDHC2_DQS
IOMUXC_PTF25_EXT_AUD_MCLK2
IOMUXC_PTF25_EPDC0_GDSP
IOMUXC_PTF25_DPI0_VSYNC
IOMUXC_PTF25_WUU1_P13
IOMUXC_PTF26_PTF26
IOMUXC_PTF26_FXIO1_D26
IOMUXC_PTF26_SPDIF_IN3
IOMUXC_PTF26_TPM7_CLKIN
IOMUXC_PTF26_I2S5_TX_BCLK
IOMUXC_PTF26_SDHC2_RESET_B
IOMUXC_PTF26_EPDC0_SDLE

IOMUXC_PTF26_DPI0_HSYNC
IOMUXC_PTF26_WUU1_P14
IOMUXC_PTF27_PTF27
IOMUXC_PTF27_FXIO1_D27
IOMUXC_PTF27_SPDIF_PLOCK
IOMUXC_PTF27_TPM7_CH0
IOMUXC_PTF27_I2S5_TX_FS
IOMUXC_PTF27_SDHC2_WP
IOMUXC_PTF27_EPDC0_SDCE0
IOMUXC_PTF27_DPI0_DE
IOMUXC_PTF27_WUU1_P15
IOMUXC_PTF28_PTF28
IOMUXC_PTF28_FXIO1_D28
IOMUXC_PTF28_SPDIF_IN4
IOMUXC_PTF28_TPM7_CH1
IOMUXC_PTF28_I2S5_TXD0
IOMUXC_PTF28_SDHC2_CD
IOMUXC_PTF28_EPDC0_SDCLK_B
IOMUXC_PTF28_LP_HV_DBG_MUX_20
IOMUXC_PTF29_PTF29
IOMUXC_PTF29_FXIO1_D29
IOMUXC_PTF29_SPDIF_SRCLK
IOMUXC_PTF29_TPM7_CH2
IOMUXC_PTF29_I2S5_TXD1
IOMUXC_PTF29_SDHC2_VS
IOMUXC_PTF29_EPDC0_SDCE1
IOMUXC_PTF29_WDOG3_RST
IOMUXC_PTF29_LP_HV_DBG_MUX_21
IOMUXC_PTF30_PTF30
IOMUXC_PTF30_FXIO1_D30
IOMUXC_PTF30_TPM7_CH3
IOMUXC_PTF30_I2S5_TXD2
IOMUXC_PTF30_MQS1_LEFT

IOMUXC_PTF30_EPDC0_SDCE2
IOMUXC_PTF30_WDOG4_RST
IOMUXC_PTF30_LP_HV_DBG_MUX_22
IOMUXC_PTF31_PTF31
IOMUXC_PTF31_FXIO1_D31
IOMUXC_PTF31_TPM7_CH4
IOMUXC_PTF31_I2S5_TXD3
IOMUXC_PTF31_MQS1_RIGHT
IOMUXC_PTF31_EPDC0_SDCE3
IOMUXC_PTF31_WDOG5_RST
IOMUXC_PTF31_LP_HV_DBG_MUX_23
IOMUXC_BOOT_MODE0_BOOT_MODE0
IOMUXC_BOOT_MODE1_BOOT_MODE1
IOMUXC_PCR_PS_MASK
IOMUXC_PCR_PS_SHIFT
IOMUXC_PCR_PS(x)
IOMUXC_PCR_PE_MASK
IOMUXC_PCR_PE_SHIFT
IOMUXC_PCR_PE(x)
IOMUXC_PCR_SRE_MASK
IOMUXC_PCR_SRE_SHIFT
IOMUXC_PCR_SRE(x)
IOMUXC_PCR_ODE_MASK
IOMUXC_PCR_ODE_SHIFT
IOMUXC_PCR_ODE(x)
IOMUXC_PCR_DSE_MASK
IOMUXC_PCR_DSE_SHIFT
IOMUXC_PCR_DSE(x)
IOMUXC_PCR_MUX_MODE_MASK
IOMUXC_PCR_MUX_MODE_SHIFT
IOMUXC_PCR_MUX_MODE(x)
IOMUXC_PCR_LK_MASK
IOMUXC_PCR_LK_SHIFT

IOMUXC_PCR_LK(x)
IOMUXC_PCR_IBE_MASK
IOMUXC_PCR_IBE_SHIFT
IOMUXC_PCR_IBE(x)
IOMUXC_PCR_OBE_MASK
IOMUXC_PCR_OBE_SHIFT
IOMUXC_PCR_OBE(x)
IOMUXC_PCR_DFE_MASK
IOMUXC_PCR_DFE_SHIFT
IOMUXC_PCR_DFE(x)
IOMUXC_PCR_DFCS_MASK
IOMUXC_PCR_DFCS_SHIFT
IOMUXC_PCR_DFCS(x)
IOMUXC_PCR_DFD_MASK
IOMUXC_PCR_DFD_SHIFT
IOMUXC_PCR_DFD(x)
IOMUXC_PSMI_SSS_MASK
IOMUXC_PSMI_SSS_SHIFT
IOMUXC_PSMI_SSS(x)
IOMUXC_PSMI_INV_MASK
IOMUXC_PSMI_INV_SHIFT
IOMUXC_PSMI_INV(x)

2.34 ISI: Image Sensing Interface

void ISI_Init(ISI_Type *base)

Initializes the ISI peripheral.

This function ungates the ISI clock, it should be called before any other ISI functions.

Parameters

- base – ISI peripheral base address.

void ISI_Deinit(ISI_Type *base)

Deinitializes the ISI peripheral.

This function gates the ISI clock.

Parameters

- base – ISI peripheral base address.


```
void ISI_Reset(ISI_Type *base)
```

Reset the ISI peripheral.

This function resets the ISI channel processing pipeline similar to a hardware reset. The channel will need to be reconfigured after reset before it can be used.

Parameters

- base – ISI peripheral base address.

```
static inline uint32_t ISI_EnableInterrupts(ISI_Type *base, uint32_t mask)
```

Enables ISI interrupts.

Parameters

- base – ISI peripheral base address
- mask – Interrupt source, OR'ed value of `_isi_interrupt`.

Returns

OR'ed value of the enabled interrupts before calling this function.

```
static inline uint32_t ISI_DisableInterrupts(ISI_Type *base, uint32_t mask)
```

Disables ISI interrupts.

Parameters

- base – ISI peripheral base address
- mask – Interrupt source, OR'ed value of `_isi_interrupt`.

Returns

OR'ed value of the enabled interrupts before calling this function.

```
static inline uint32_t ISI_GetInterruptStatus(ISI_Type *base)
```

Get the ISI interrupt pending flags.

All interrupt pending flags are returned, upper layer could compare with the OR'ed value of `_isi_interrupt`. For example, to check whether memory read completed, use like this:

```
uint32_t mask = ISI_GetInterruptStatus(ISI);
if (mask & kISI_MemReadCompletedInterrupt)
{
    memory read completed
}
```

Parameters

- base – ISI peripheral base address

Returns

The OR'ed value of the pending interrupt flags. of `_isi_interrupt`.

```
static inline void ISI_ClearInterruptStatus(ISI_Type *base, uint32_t mask)
```

Clear ISI interrupt pending flags.

This function could clear one or more flags at one time, the flags to clear are passed in as an OR'ed value of `_isi_interrupt`. For example, to clear both line received interrupt flag and frame received flag, use like this:

```
ISI_ClearInterruptStatus(ISI, kISI_LineReceivedInterrupt | kISI_FrameReceivedInterrupt);
```

Parameters

- base – ISI peripheral base address
- mask – The flags to clear, it is OR'ed value of `_isi_interrupt`.

```
void ISI_SetScalerConfig(ISI_Type *base, uint16_t inputWidth, uint16_t inputHeight, uint16_t outputWidth, uint16_t outputHeight)
```

Set the ISI channel scaler configurations.

This function sets the scaling configurations. If the ISI channel is bypassed, then the scaling feature could not be used.

ISI only supports down scaling but not up scaling.

Note: Total bytes in one line after down scaling must be more than 256 bytes.

Parameters

- base – ISI peripheral base address
- inputWidth – Input image width.
- inputHeight – Input image height.
- outputWidth – Output image width.
- outputHeight – Output image height.

```
void ISI_SetColorSpaceConversionConfig(ISI_Type *base, const isi_csc_config_t *config)
```

Set the ISI color space conversion configurations.

This function sets the color space conversion configurations. After setting the configuration, use the function `ISI_EnableColorSpaceConversion` to enable this feature. If the ISI channel is bypassed, then the color space conversion feature could not be used.

Parameters

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

```
void ISI_ColorSpaceConversionGetDefaultConfig(isi_csc_config_t *config)
```

Get the ISI color space conversion default configurations.

The default value is:

```
config->mode = kISI_CscYUV2RGB;
config->A1 = 0.0;
config->A2 = 0.0;
config->A3 = 0.0;
config->B1 = 0.0;
config->B2 = 0.0;
config->B3 = 0.0;
config->C1 = 0.0;
config->C2 = 0.0;
config->C3 = 0.0;
config->D1 = 0;
config->D2 = 0;
config->D3 = 0;
```

Parameters

- config – Pointer to the configuration structure.

```
static inline void ISI_EnableColorSpaceConversion(ISI_Type *base, bool enable)
```

Enable or disable the ISI color space conversion.

If the ISI channel is bypassed, then the color space conversion feature could not be used even enable using this function.

Note: The CSC is enabled by default. Disable it if it is not required.

Parameters

- base – ISI peripheral base address
- enable – True to enable, false to disable.

void ISI_SetCropConfig(ISI_Type *base, const *isi_crop_config_t* *config)

Set the ISI cropping configurations.

This function sets the cropping configurations. After setting the configuration, use the function ISI_EnableCrop to enable the feature. Cropping still works when the ISI channel is bypassed.

Note: The upper left corner and lower right corner should be configured base on the image resolution output from the scaler.

Parameters

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

void ISI_CropGetDefaultConfig(*isi_crop_config_t* *config)

Get the ISI cropping default configurations.

The default value is:

```
config->upperLeftX = 0U;
config->upperLeftY = 0U;
config->lowerRightX = 0U;
config->lowerRightY = 0U;
```

Parameters

- config – Pointer to the configuration structure.

static inline void ISI_EnableCrop(ISI_Type *base, bool enable)

Enable or disable the ISI cropping.

If the ISI channel is bypassed, the cropping still works.

Parameters

- base – ISI peripheral base address
- enable – True to enable, false to disable.

static inline void ISI_SetGlobalAlpha(ISI_Type *base, uint8_t alpha)

Set the global alpha value.

Parameters

- base – ISI peripheral base address
- alpha – The global alpha value.

static inline void ISI_EnableGlobalAlpha(ISI_Type *base, bool enable)

Enable the global alpha insertion.

Alpha still works when channel bypassed.

Parameters

- base – ISI peripheral base address
- enable – True to enable, false to disable.

```
void ISI_SetRegionAlphaConfig(ISI_Type *base, uint8_t index, const isi_region_alpha_config_t *config)
```

Set the alpha value for region of interest.

Set the alpha insertion configuration for specific region of interest. The function `ISI_EnableRegionAlpha` could be used to enable the alpha insertion. Alpha insertion still works when channel bypassed.

Note: The upper left corner and lower right corner should be configured base on the image resolution output from the scaler.

Parameters

- base – ISI peripheral base address
- index – Index of the region of interest, Could be 0, 1, 2, and 3.
- config – Pointer to the configuration structure.

```
void ISI_RegionAlphaGetDefaultConfig(isi_region_alpha_config_t *config)
```

Get the regional alpha insertion default configurations.

The default configuration is:

```
config->upperLeftX = 0U;  
config->upperLeftY = 0U;  
config->lowerRightX = 0U;  
config->lowerRightY = 0U;  
config->alpha = 0U;
```

Parameters

- config – Pointer to the configuration structure.

```
void ISI_EnableRegionAlpha(ISI_Type *base, uint8_t index, bool enable)
```

Enable or disable the alpha value insertion for region of interest.

Alpha insertion still works when channel bypassed.

Parameters

- base – ISI peripheral base address
- index – Index of the region of interest, Could be 0, 1, 2, and 3.
- enable – True to enable, false to disable.

```
void ISI_SetInputMemConfig(ISI_Type *base, const isi_input_mem_config_t *config)
```

Set the input memory configuration.

Parameters

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

```
void ISI_InputMemGetDefaultConfig(isi_input_mem_config_t *config)
```

Get the input memory default configurations.

The default configuration is:

```
config->addr = 0U;
config->linePitchBytes = 0U;
config->framePitchBytes = 0U;
config->format = kISI_InputMemBGR8P;
```

Parameters

- config – Pointer to the configuration structure.

static inline void ISI_SetInputMemAddr(ISI_Type *base, uint32_t addr)

Set the input memory address.

This function only sets the input memory address, it is used for fast run-time setting.

Parameters

- base – ISI peripheral base address
- addr – Input memory address.

void ISI_TriggerInputMemRead(ISI_Type *base)

Trigger the ISI pipeline to read the input memory.

Parameters

- base – ISI peripheral base address

static inline void ISI_SetFlipMode(ISI_Type *base, *isi_flip_mode_t* mode)

Set the ISI channel flipping mode.

Parameters

- base – ISI peripheral base address
- mode – Flipping mode.

void ISI_SetOutputBufferAddr(ISI_Type *base, uint8_t index, uint32_t addrY, uint32_t addrU, uint32_t addrV)

Set the ISI output buffer address.

This function sets the output buffer address and trigger the ISI to shadow the address, it is used for fast run-time setting.

Parameters

- base – ISI peripheral base address
- index – Index of output buffer, could be 0 and 1.
- addrY – RGB or Luma (Y) output buffer address.
- addrU – Chroma (U/Cb/UV/CbCr) output buffer address.
- addrV – Chroma (V/Cr) output buffer address.

static inline void ISI_Start(ISI_Type *base)

Start the ISI channel.

Start the ISI channel to work, this function should be called after all channel configuration finished.

Parameters

- base – ISI peripheral base address

static inline void ISI_Stop(ISI_Type *base)

Stop the ISI channel.

Parameters

- base – ISI peripheral base address

FSL_ISI_DRIVER_VERSION

ISI driver version.

enum __isi_interrupt

ISI interrupts.

Values:

enumerator kISI_MemReadCompletedInterrupt

Input memory read completed.

enumerator kISI_LineReceivedInterrupt

Line received.

enumerator kISI_FrameReceivedInterrupt

Frame received.

enumerator kISI_AxiWriteErrorVInterrupt

AXI Bus write error when storing V data to memory.

enumerator kISI_AxiWriteErrorUInterrupt

AXI Bus write error when storing U data to memory.

enumerator kISI_AxiWriteErrorYInterrupt

AXI Bus write error when storing Y data to memory.

enumerator kISI_AxiReadErrorInterrupt

AXI Bus error when reading the input memory.

enumerator kISI_OverflowAlertVInterrupt

V output buffer overflow threshold accrossed.

enumerator kISI_ExcessOverflowVInterrupt

V output buffer excess overflow interrupt.

enumerator kISI_OverflowVInterrupt

V output buffer overflow interrupt.

enumerator kISI_OverflowAlertUInterrupt

U output buffer overflow threshold accrossed.

enumerator kISI_ExcessOverflowUInterrupt

U output buffer excess overflow interrupt.

enumerator kISI_OverflowUInterrupt

U output buffer overflow interrupt.

enumerator kISI_OverflowAlertYInterrupt

V output buffer overflow threshold accrossed.

enumerator kISI_ExcessOverflowYInterrupt

V output buffer excess overflow interrupt.

enumerator kISI_OverflowYInterrupt

V output buffer overflow interrupt.

enum __isi_output_format

ISI output image format.

Values:

enumerator kISI_OutputRGBA8888
 RGBA8888.

enumerator kISI_OutputABGR8888
 ABGR8888.

enumerator kISI_OutputARGB8888
 ARGB8888.

enumerator kISI_OutputRGBX8888
 RGBX8888 unpacked and MSB aligned in 32-bit.

enumerator kISI_OutputXBGR8888
 XBGR8888 unpacked and LSB aligned in 32-bit.

enumerator kISI_OutputXRGB8888
 XRGB8888 unpacked and LSB aligned in 32-bit.

enumerator kISI_OutputRGB888
 RGB888 packed into 32-bit.

enumerator kISI_OutputBGR888
 BGR888 packed into 32-bit.

enumerator kISI_OutputA2BGR10
 BGR format with 2-bits alpha in MSB; 10-bits per color component.

enumerator kISI_OutputA2RGB10
 RGB format with 2-bits alpha in MSB; 10-bits per color component.

enumerator kISI_OutputRGB565
 RGB565 packed into 32-bit.

enumerator kISI_OutputRaw8
 8-bit raw data packed into 32-bit.

enumerator kISI_OutputRaw10
 10-bit raw data packed into 16-bit with 6 LSBs wasted.

enumerator kISI_OutputRaw10P
 10-bit raw data packed into 32-bit.

enumerator kISI_OutputRaw12P
 16-bit raw data packed into 16-bit with 4 LSBs wasted.

enumerator kISI_OutputRaw16P
 16-bit raw data packed into 32-bit.

enumerator kISI_OutputYUV444_1P8P
 8-bits per color component; 1-plane, YUV interleaved packed bytes.

enumerator kISI_OutputYUV444_2P8P
 8-bits per color component; 2-plane, UV interleaved packed bytes.

enumerator kISI_OutputYUV444_3P8P
 8-bits per color component; 3-plane, non-interleaved packed bytes.

enumerator kISI_OutputYUV444_1P8
 8-bits per color component; 1-plane YUV interleaved unpacked bytes (8 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV444_1P10

10-bits per color component; 1-plane, YUV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV444_2P10

10-bits per color component; 2-plane, UV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV444_3P10

10-bits per color component; 3-plane, non-interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV444_1P10P

10-bits per color component; 1-plane, YUV interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV444_2P10P

10-bits per color component; 2-plane, UV interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV444_3P10P

10-bits per color component; 3-plane, non-interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV444_1P12

12-bits per color component; 1-plane, YUV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV444_2P12

12-bits per color component; 2-plane, UV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV444_3P12

12-bits per color component; 3-plane, non-interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_1P8P

8-bits per color component; 1-plane, YUV interleaved packed bytes.

enumerator kISI_OutputYUV422_2P8P

8-bits per color component; 2-plane, UV interleaved packed bytes.

enumerator kISI_OutputYUV422_3P8P

8-bits per color component; 3-plane, non-interleaved packed bytes.

enumerator kISI_OutputYUV422_1P10

10-bits per color component; 1-plane, YUV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_2P10

10-bits per color component; 2-plane, UV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_3P10

10-bits per color component; 3-plane, non-interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_1P10P

10-bits per color component; 1-plane, YUV interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV422_2P10P

10-bits per color component; 2-plane, UV interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV422_3P10P

10-bits per color component; 3-plane, non-interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV422_1P12

12-bits per color component; 1-plane, YUV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_2P12

12-bits per color component; 2-plane, UV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV422_3P12

12-bits per color component; 3-plane, non-interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV420_2P8P

8-bits per color component; 2-plane, UV interleaved packed bytes.

enumerator kISI_OutputYUV420_3P8P

8-bits per color component; 3-plane, non-interleaved packed bytes.

enumerator kISI_OutputYUV420_2P10

10-bits per color component; 2-plane, UV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV420_3P10

10-bits per color component; 3-plane, non-interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV420_2P10P

10-bits per color component; 2-plane, UV interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV420_3P10P

10-bits per color component; 3-plane, non-interleaved packed bytes (2 MSBs waste bits in 32-bit DWORD).

enumerator kISI_OutputYUV420_2P12

12-bits per color component; 2-plane, UV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_OutputYUV420_3P12

12-bits per color component; 3-plane, non-interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enum _isi_chain_mode

ISI line buffer chain mode.

Values:

enumerator kISI_ChainDisable

No line buffers chained, for 2048 or less horizontal resolution.

enumerator kISI_ChainTwo

Line buffers of channel n and n+1 chained, for 4096 horizontal resolution.

enum _isi_deint_mode

ISI de-interlacing mode.

Values:

enumerator kISI_DeintDisable

No de-interlacing.

enumerator kISI_DeintWeaveOddOnTop

Weave de-interlacing (Odd, Even) method used.

enumerator kISI_DeintWeaveEvenOnTop

Weave de-interlacing (Even, Odd) method used.

enumerator kISI_DeintBlendingOddFirst

Blending or linear interpolation (Odd + Even).

enumerator kISI_DeintBlendingEvenFirst

Blending or linear interpolation (Even + Odd).

enumerator kISI_DeintDoublingOdd

Doubling odd frame and discard even frame.

enumerator kISI_DeintDoublingEven

Doubling even frame and discard odd frame.

enum _isi_threshold

ISI overflow panic alert threshold.

Values:

enumerator kISI_ThresholdDisable

No panic alert will be asserted.

enumerator kISI_Threshold25Percent

Panic will assert when the buffers are 25% full.

enumerator kISI_Threshold50Percent

Panic will assert when the buffers are 50% full.

enumerator kISI_Threshold75Percent

Panic will assert when the buffers are 75% full.

enum _isi_csc_mode

ISI color space conversion mode.

Values:

enumerator kISI_CscYUV2RGB

Convert YUV to RGB.

enumerator kISI_CscYCbCr2RGB

Convert YCbCr to RGB.

enumerator kISI_CscRGB2YUV

Convert RGB to YUV.

enumerator kISI_CscRGB2YCbCr

Convert RGB to YCbCr.

enum _isi_flip_mode

ISI flipping mode.

Values:

enumerator kISI_FlipDisable

Flip disabled.

enumerator kISI_FlipHorizontal

Horizontal flip.

enumerator kISI_FlipVertical

Vertical flip.

enumerator kISI_FlipBoth

Flip both direction.

enum _isi_input_mem_format

ISI image format of the input memory.

Values:

enumerator kISI_InputMemBGR888

BGR format with 8-bits per color component, packed into 32-bit, 24 bits per pixel.

enumerator kISI_InputMemRGB888

RGB format with 8-bits per color component, packed into 32-bit, 24 bits per pixel.

enumerator kISI_InputMemXRGB8888

RGB format with 8-bits per color component, unpacked and LSB aligned in 32-bit, 32 bits per pixel.

enumerator kISI_InputMemRGBX8888

RGB format with 8-bits per color component, unpacked and MSB aligned in 32-bit, 32 bits per pixel.

enumerator kISI_InputMemXBGR8888

BGR format with 8-bits per color component, unpacked and LSB aligned in 32-bit, 32 bits per pixel.

enumerator kISI_InputMemRGB565

RGB format with 5-bits of R, B; 6-bits of G (packed into 32-bit)

enumerator kISI_InputMemA2BGR10

BGR format with 2-bits alpha in MSB; 10-bits per color component.

enumerator kISI_InputMemA2RGB10

RGB format with 2-bits alpha in MSB; 10-bits per color component.

enumerator kISI_InputMemYUV444_1P8P

8-bits per color component; 1-plane, YUV interleaved packed bytes.

enumerator kISI_InputMemYUV444_1P10

10-bits per color component; 1-plane, YUV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_InputMemYUV444_1P10P

10-bits per color component; 1-plane, YUV interleaved packed bytes (2 MSBs waste bits in 32-bit WORD).

enumerator kISI_InputMemYUV444_1P12

12-bits per color component; 1-plane, YUV interleaved unpacked bytes (4 LSBs waste bits in 16-bit WORD).

enumerator kISI_InputMemYUV444_1P8

8-bits per color component; 1-plane YUV interleaved unpacked bytes (8 MSBs waste bits in 32-bit DWORD).

enumerator kISI_InputMemYUV422_1P8P

8-bits per color component; 1-plane YUV interleaved packed bytes.

enumerator kISI_InputMemYUV422_1P10

10-bits per color component; 1-plane, YUV interleaved unpacked bytes (6 LSBs waste bits in 16-bit WORD).

enumerator kISI_InputMemYUV422_1P12

12-bits per color component; 1-plane, YUV interleaved packed bytes (4 MSBs waste bits in 16-bit WORD).

typedef enum *_isi_output_format* isi_output_format_t

ISI output image format.

typedef enum *_isi_chain_mode* isi_chain_mode_t

ISI line buffer chain mode.

typedef enum *_isi_deint_mode* isi_deint_mode_t

ISI de-interlacing mode.

typedef enum *_isi_threshold* isi_threshold_t

ISI overflow panic alert threshold.

typedef struct *_isi_config* isi_config_t

ISI basic configuration.

typedef enum *_isi_csc_mode* isi_csc_mode_t

ISI color space conversion mode.

typedef struct *_isi_csc_config* isi_csc_config_t

ISI color space conversion configurations.

(a) RGB to YUV (or YCbCr) conversion

- $Y = (A1 \times R) + (A2 \times G) + (A3 \times B) + D1$
- $U = (B1 \times R) + (B2 \times G) + (B3 \times B) + D2$
- $V = (C1 \times R) + (C2 \times G) + (C3 \times B) + D3$

(b) YUV (or YCbCr) to RGB conversion

- $R = (A1 \times (Y + D1)) + (A2 \times (U + D2)) + (A3 \times (V + D3))$
- $G = (B1 \times (Y + D1)) + (B2 \times (U + D2)) + (B3 \times (V + D3))$
- $B = (C1 \times (Y + D1)) + (C2 \times (U + D2)) + (C3 \times (V + D3))$

Overflow for the three channels are saturated at 0x255 and underflow is saturated at 0x00.

typedef enum *_isi_flip_mode* isi_flip_mode_t

ISI flipping mode.

typedef struct *_isi_crop_config* isi_crop_config_t

ISI cropping configurations.

typedef struct *_isi_regoi_alpha_config* isi_region_alpha_config_t

ISI regional region alpha configurations.

typedef enum *_isi_input_mem_format* isi_input_mem_format_t

ISI image format of the input memory.

typedef struct *_isi_input_mem_config* isi_input_mem_config_t

ISI input memory configurations.

```
void ISI_SetConfig(ISI_Type *base, const isi_config_t *config)
```

Set the ISI channel basic configurations.

This function sets the basic configurations, generally the channel could be started to work after this function. To enable other features such as cropping, flipping, please call the functions accordingly.

Parameters

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

```
void ISI_GetDefaultConfig(isi_config_t *config)
```

Get the ISI channel default basic configurations.

The default value is:

```
config->isChannelBypassed = false;
config->isSourceMemory = false;
config->isYCbCr = false;
config->chainMode = kISI_ChainDisable;
config->deintMode = kISI_DeintDisable;
config->blankPixel = 0xFFU;
config->sourcePort = 0U;
config->mipiChannel = 0U;
config->inputHeight = 1080U;
config->inputWidth = 1920U;
config->outputFormat = kISI_OutputRGBA8888;
config->outputLinePitchBytes = 0U;
config->thresholdY = kISI_ThresholdDisable;
config->thresholdU = kISI_ThresholdDisable;
config->thresholdV = kISI_ThresholdDisable;
```

Parameters

- config – Pointer to the configuration structure.

```
struct __isi_config
```

#include <fsl_isi.h> ISI basic configuration.

Public Members

```
bool isChannelBypassed
```

Bypass the channel, if bypassed, the scaling and color space conversion could not work.

```
bool isSourceMemory
```

Whether the input source is memory or not.

```
bool isYCbCr
```

Whether the input source is YCbCr mode or not.

```
isi_chain_mode_t chainMode
```

The line buffer chain mode.

```
isi_deint_mode_t deintMode
```

The de-interlacing mode.

```
uint8_t blankPixel
```

The pixel to insert into image when overflow occurs.

```
uint8_t sourcePort
```

Input source port selection.

uint8_t mipiChannel

MIPI virtual channel, ignored if input source is not MIPI CSI.

uint16_t inputHeight

Input image height(lines).

uint16_t inputWidth

Input image width(pixels).

isi_output_format_t outputFormat

Output image format.

isi_threshold_t thresholdY

Panic alert threshold for RGB or Luma (Y) buffer.

isi_threshold_t thresholdU

Panic alert threshold for Chroma (U/Cb/UV/CbCr) buffer.

isi_threshold_t thresholdV

Panic alert threshold for Chroma (V/Cr) buffer.

struct __isi_csc_config

#include <fsl_isi.h> ISI color space conversion configurations.

(a) RGB to YUV (or YCbCr) conversion

- $Y = (A1 \times R) + (A2 \times G) + (A3 \times B) + D1$
- $U = (B1 \times R) + (B2 \times G) + (B3 \times B) + D2$
- $V = (C1 \times R) + (C2 \times G) + (C3 \times B) + D3$

(b) YUV (or YCbCr) to RGB conversion

- $R = (A1 \times (Y + D1)) + (A2 \times (U + D2)) + (A3 \times (V + D3))$
- $G = (B1 \times (Y + D1)) + (B2 \times (U + D2)) + (B3 \times (V + D3))$
- $B = (C1 \times (Y + D1)) + (C2 \times (U + D2)) + (C3 \times (V + D3))$

Overflow for the three channels are saturated at 0x255 and underflow is saturated at 0x00.

Public Members

isi_csc_mode_t mode

Conversion mode.

float A1

Must be in the range of [-3.99609375, 3.99609375].

float A2

Must be in the range of [-3.99609375, 3.99609375].

float A3

Must be in the range of [-3.99609375, 3.99609375].

float B1

Must be in the range of [-3.99609375, 3.99609375].

float B2

Must be in the range of [-3.99609375, 3.99609375].

float B3

Must be in the range of [-3.99609375, 3.99609375].

float C1

Must be in the range of [-3.99609375, 3.99609375].

float C2

Must be in the range of [-3.99609375, 3.99609375].

float C3

Must be in the range of [-3.99609375, 3.99609375].

int32_t D1

Must be in the range of [-256, 255].

int32_t D2

Must be in the range of [-256, 255].

int32_t D3

Must be in the range of [-256, 255].

struct __isi_crop_config

#include <fsl_isi.h> ISI cropping configurations.

Public Members

uint16_t upperLeftX

X of upper left corner.

uint16_t upperLeftY

Y of upper left corner.

uint16_t lowerRightX

X of lower right corner.

uint16_t lowerRightY

Y of lower right corner.

struct __isi_regoin_alpha_config

#include <fsl_isi.h> ISI regional region alpha configurations.

Public Members

uint16_t upperLeftX

X of upper left corner.

uint16_t upperLeftY

Y of upper left corner.

uint16_t lowerRightX

X of lower right corner.

uint16_t lowerRightY

Y of lower right corner.

uint8_t alpha

Alpha value.

struct __isi_input_mem_config

#include <fsl_isi.h> ISI input memory configurations.

Public Members

`uint32_t addr`
Address of the input memory.

`uint16_t linePitchBytes`
Line pitch in bytes.

`uint16_t framePitchBytes`
Frame pitch in bytes.

`isi_input_mem_format_t format`
Image format of the input memory.

2.35 Common Driver

`FSL_COMMON_DRIVER_VERSION`
common driver version.

`DEBUG_CONSOLE_DEVICE_TYPE_NONE`
No debug console.

`DEBUG_CONSOLE_DEVICE_TYPE_UART`
Debug console based on UART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPUART`
Debug console based on LPUART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPSCI`
Debug console based on LPSCI.

`DEBUG_CONSOLE_DEVICE_TYPE_USBCDC`
Debug console based on USBCDC.

`DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM`
Debug console based on FLEXCOMM.

`DEBUG_CONSOLE_DEVICE_TYPE_IUART`
Debug console based on i.MX UART.

`DEBUG_CONSOLE_DEVICE_TYPE_VUSART`
Debug console based on LPC_VUSART.

`DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART`
Debug console based on LPC_USART.

`DEBUG_CONSOLE_DEVICE_TYPE_SWO`
Debug console based on SWO.

`DEBUG_CONSOLE_DEVICE_TYPE_QSCI`
Debug console based on QSCI.

`MIN(a, b)`
Computes the minimum of *a* and *b*.

`MAX(a, b)`
Computes the maximum of *a* and *b*.

`UINT16_MAX`
Max value of `uint16_t` type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true**, else return **false**.

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_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

CACHE_LINE_DATA

enum __status_groups

Status group numbers.

Values:

enumerator kStatusGroup_Generic

Group number for generic status codes.

enumerator kStatusGroup_FLASH

Group number for FLASH status codes.

enumerator kStatusGroup_LPSPi
Group number for LPSPi status codes.

enumerator kStatusGroup_FLEXIO_SPI
Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI
Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
Group number for LPI2C status codes.

enumerator kStatusGroup_UART
Group number for UART status codes.

enumerator kStatusGroup_I2C
Group number for UART status codes.

enumerator kStatusGroup_LPSCI
Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
Group number for LPUART status codes.

enumerator kStatusGroup_SPI
Group number for SPI status code.

enumerator kStatusGroup_XRDC
Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPi
Group number for ECSPi status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
Group number for debug console status codes.

enumerator kStatusGroup_SEMC
Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
Starting number for application groups.

enumerator kStatusGroup_IAP
Group number for IAP status codes

enumerator kStatusGroup_SFA
Group number for SFA status codes

enumerator kStatusGroup_SPC
Group number for SPC status codes.

enumerator kStatusGroup_PUF
Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes

enumerator kStatusGroup_VBAT
Group number for VBAT status codes

enumerator kStatusGroup_XSPI
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

enumerator kStatusGroup_OSA
Group number for OSA status codes.

enumerator kStatusGroup_COMMON_TASK
Group number for Common task status codes.

enumerator kStatusGroup_MSG
Group number for messaging status codes.

enumerator kStatusGroup_SDK_OCOTP
Group number for OCOTP status codes.

enumerator kStatusGroup_SDK_FLEXSPINOR
Group number for FLEXSPINOR status codes.

enumerator kStatusGroup_CODEC
Group number for codec status codes.

enumerator kStatusGroup_ASRC
Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
Group number for codec status codes.

enumerator kStatusGroup_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_ELE MU
Group number for ELE MU status codes.

enumerator kStatusGroup_QUEUE DSPI
Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3

Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE

Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX

Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC

Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT

Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT

Group number for A-format status codes.

Generic status return codes.

Values:

enumerator kStatus_Success

Generic status for Success.

enumerator kStatus_Fail

Generic status for Fail.

enumerator kStatus_ReadOnly

Generic status for read only failure.

enumerator kStatus_OutOfRange

Generic status for out of range access.

enumerator kStatus_InvalidArgument

Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

`void SDK_Free(void *ptr)`

Free memory.

Parameters

- `ptr` – The memory to be release.

`void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)`

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- `delayTime_us` – Delay time in unit of microsecond.
- `coreClock_Hz` – Core clock frequency with Hz.

`static inline status_t EnableIRQ(IRQn_Type interrupt)`

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt enabled successfully
- `kStatus_Fail` – Failed to enable the interrupt

`static inline status_t DisableIRQ(IRQn_Type interrupt)`

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt disabled successfully
- `kStatus_Fail` – Failed to disable the interrupt

`static inline status_t EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)`

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management

mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

```
static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t
                                                    newValue)
```

```
static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)
```

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version		Minor Version		Bug Fix		
31	25	24	17	16	9	8	0

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING();” need to be added at the end of each case section which misses “break;”statement.

MSDK_REG_SECURE_ADDR(x)

Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

Convert the register address to the one used in non-secure mode.

MSDK_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.36 LCDIF: LCD interface

```
status_t LCDIF_Init(LCDIF_Type *base)
```

Initialize the LCDIF.

This function initializes the LCDIF to work.

Parameters

- base – LCDIF peripheral base address.

Return values

kStatus__Success – Initialize successfully.

void LCDIF_Deinit(LCDIF_Type *base)

De-initialize the LCDIF.

This function disables the LCDIF peripheral clock.

Parameters

- base – LCDIF peripheral base address.

void LCDIF_DpiModeGetDefaultConfig(*lcdif_dpi_config_t* *config)

Get the default configuration for to initialize the LCDIF.

The default configuration value is:

```
config->panelWidth = 0;
config->panelHeight = 0;
config->hsw = 0;
config->hfp = 0;
config->hbp = 0;
config->vsw = 0;
config->vfp = 0;
config->vbp = 0;
config->polarityFlags = kLCDIF_VsyncActiveLow | kLCDIF_HsyncActiveLow | kLCDIF_
↪DataEnableActiveHigh |
kLCDIF_DriveDataOnFallingClkEdge; config->format = kLCDIF_Output24Bit;
```

Parameters

- config – Pointer to the LCDIF configuration.

status_t LCDIF_DpiModeSetConfig(LCDIF_Type *base, uint8_t displayIndex, const *lcdif_dpi_config_t* *config)

Initialize the LCDIF to work in DPI mode.

This function configures the LCDIF DPI display.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- config – Pointer to the configuration structure.

Return values

- kStatus__Success – Initialize successfully.
- kStatus__InvalidArgument – Initialize failed because of invalid argument.

status_t LCDIF_DbiModeSetConfig(LCDIF_Type *base, uint8_t displayIndex, const *lcdif_dbi_config_t* *config)

Initialize the LCDIF to work in DBI mode.

This function configures the LCDIF DBI display.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- config – Pointer to the configuration structure.

Return values

- kStatus__Success – Initialize successfully.

- `kStatus_InvalidArgument` – Initialize failed because of invalid argument.

`void LCDIF_DbiModeGetDefaultConfig(lcdif_dbi_config_t *config)`

Get the default configuration to initialize the LCDIF DBI mode.

The default configuration value is:

```
config->swizzle      = kLCDIF_DbiOutSwizzleRGB;
config->format        = kLCDIF_DbiOutD8RGB332;
config->acTimeUnit    = 0;
config->type          = kLCDIF_DbiTypeA_ClockedE;
config->reversePolarity = false;
config->writeWRPeriod  = 3U;
config->writeWRAssert  = 0U;
config->writeCSAssert  = 0U;
config->writeWRDeassert = 0U;
config->writeCSDeassert = 0U;
config->typeCTas       = 1U;
config->typeCSCLTwrl   = 1U;
config->typeCSCLTwrh   = 1U;
```

Parameters

- `config` – Pointer to the LCDIF DBI configuration.

`static inline void LCDIF_DbiReset(LCDIF_Type *base, uint8_t displayIndex)`

Reset the DBI module.

Parameters

- `displayIndex` – Display index.
- `base` – LCDIF peripheral base address.

`static inline bool LCDIF_DbiIsTypeCFifoFull(LCDIF_Type *base, uint8_t displayIndex)`

Check whether the FIFO is full in DBI mode type C.

Parameters

- `base` – LCDIF peripheral base address.
- `displayIndex` – Display index.

Return values

- `true` – FIFO full.
- `false` – FIFO not full.

`void LCDIF_DbiSelectArea(LCDIF_Type *base, uint8_t displayIndex, uint16_t startX, uint16_t startY, uint16_t endX, uint16_t endY, bool isTiled)`

Select the update area in DBI mode.

Parameters

- `base` – LCDIF peripheral base address.
- `displayIndex` – Display index.
- `startX` – X coordinate for start pixel.
- `startY` – Y coordinate for start pixel.
- `endX` – X coordinate for end pixel.
- `endY` – Y coordinate for end pixel.
- `isTiled` – true if the pixel data is tiled.

```
static inline void LCDIF_DbiSendCommand(LCDIF_Type *base, uint8_t displayIndex, uint8_t cmd)
```

Send command to DBI port.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- cmd – the DBI command to send.

```
void LCDIF_DbiSendData(LCDIF_Type *base, uint8_t displayIndex, const uint8_t *data, uint32_t dataLen_Byte)
```

brief Send data to DBI port.

Can be used to send light weight data to panel. To send pixel data in frame buffer, use LCDIF_DbiWriteMem.

param base LCDIF peripheral base address. param displayIndex Display index. param data pointer to data buffer. param dataLen_Byte data buffer length in byte.

```
void LCDIF_DbiSendCommandAndData(LCDIF_Type *base, uint8_t displayIndex, uint8_t cmd, const uint8_t *data, uint32_t dataLen_Byte)
```

Send command followed by data to DBI port.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- cmd – the DBI command to send.
- data – pointer to data buffer.
- dataLen_Byte – data buffer length in byte.

```
static inline void LCDIF_DbiWriteMem(LCDIF_Type *base, uint8_t displayIndex)
```

Send pixel data in frame buffer to panel controller memory.

This function starts sending the pixel data in frame buffer to panel controller, user can monitor interrupt kLCDIF_Display0FrameDoneInterrupt to know when then data sending finished.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.

```
void LCDIF_SetFrameBufferConfig(LCDIF_Type *base, uint8_t displayIndex, const lcdif_fb_config_t *config)
```

Configure the LCDIF frame buffer.

@Note: For LCDIF of version DC8000 there can be 3 layers in the pre-processing, compared with the older version. Apart from the video layer, there are also 2 overlay layers which shares the same configurations. Use this API to configure the legacy video layer, and use LCDIF_SetOverlayFrameBufferConfig to configure the overlay layers.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- config – Pointer to the configuration structure.

void LCDIF_FramerBufferGetDefaultConfig(*lcdif_fb_config_t* *config)

Get default frame buffer configuration.

@Note: For LCDIF of version DC8000 there can be 3 layers in the pre-processing, compared with the older version. Apart from the video layer, there are also 2 overlay layers which shares the same configurations. Use this API to get the default configuration for all the 3 layers.

The default configuration is

```
config->enable = true;
config->enableGamma = false;
config->format = kLCDIF_PixelFormatRGB565;
```

Parameters

- config – Pointer to the configuration structure.

static inline void LCDIF_SetFrameBufferAddr(LCDIF_Type *base, uint8_t displayIndex, uint32_t address)

Set the frame buffer to LCDIF.

Note: The address must be 128 bytes aligned.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- address – Frame buffer address.

void LCDIF_SetFrameBufferStride(LCDIF_Type *base, uint8_t displayIndex, uint32_t strideBytes)

Set the frame buffer stride.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- strideBytes – The stride in byte.

void LCDIF_SetDitherConfig(LCDIF_Type *base, uint8_t displayIndex, const *lcdif_dither_config_t* *config)

Set the dither configuration.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Index to configure.
- config – Pointer to the configuration structure.

void LCDIF_SetGammaData(LCDIF_Type *base, uint8_t displayIndex, uint16_t startIndex, const uint32_t *gamma, uint16_t gammaLen)

Set the gamma translation values to the LCDIF gamma table.

Parameters

- base – LCDIF peripheral base address.
- displayIndex – Display index.
- startIndex – Start index in the gamma table that the value will be set to.

- gamma – The gamma values to set to the gamma table in LCDIF, could be defined using LCDIF_MAKE_GAMMA_VALUE.
- gammaLen – The length of the gamma.

static inline void LCDIF_EnableInterrupts(LCDIF_Type *base, uint32_t mask)

Enables LCDIF interrupt requests.

Parameters

- base – LCDIF peripheral base address.
- mask – The interrupts to enable, pass in as OR'ed value of _lcdif_interrupt.

static inline void LCDIF_DisableInterrupts(LCDIF_Type *base, uint32_t mask)

Disable LCDIF interrupt requests.

Parameters

- base – LCDIF peripheral base address.
- mask – The interrupts to disable, pass in as OR'ed value of _lcdif_interrupt.

static inline uint32_t LCDIF_GetAndClearInterruptPendingFlags(LCDIF_Type *base)

Get and clear LCDIF interrupt pending status.

Note: The interrupt must be enabled, otherwise the interrupt flags will not assert.

Parameters

- base – LCDIF peripheral base address.

Returns

The interrupt pending status.

void LCDIF_CursorGetDefaultConfig(*lcdif_cursor_config_t* *config)

Get the hardware cursor default configuration.

The default configuration values are:

```
config->enable = true;
config->format = kLCDIF_CursorMasked;
config->hotspotOffsetX = 0;
config->hotspotOffsetY = 0;
```

Parameters

- config – Pointer to the hardware cursor configuration structure.

void LCDIF_SetCursorConfig(LCDIF_Type *base, const *lcdif_cursor_config_t* *config)

Configure the cursor.

Parameters

- base – LCDIF peripheral base address.
- config – Cursor configuration.

static inline void LCDIF_SetCursorHotspotPosition(LCDIF_Type *base, uint16_t x, uint16_t y)

Set the cursor hotspot postion.

Parameters

- base – LCDIF peripheral base address.
- x – X coordinate of the hotspot, range 0 ~ 8191.

- y – Y coordinate of the hotspot, range 0 ~ 8191.

static inline void LCDIF_SetCursorBufferAddress(LCDIF_Type *base, uint32_t address)

Set the cursor memory address.

Parameters

- base – LCDIF peripheral base address.
- address – Memory address.

void LCDIF_SetCursorColor(LCDIF_Type *base, uint32_t background, uint32_t foreground)

Set the cursor color.

Parameters

- base – LCDIF peripheral base address.
- background – Background color, could be defined use LCDIF_MAKE_CURSOR_COLOR
- foreground – Foreground color, could be defined use LCDIF_MAKE_CURSOR_COLOR

FSL_LCDIF_DRIVER_VERSION

enum _lcdif_polarity_flags

LCDIF signal polarity flags.

Values:

enumerator kLCDIF_VsyncActiveLow
VSYNC active low.

enumerator kLCDIF_VsyncActiveHigh
VSYNC active high.

enumerator kLCDIF_HsyncActiveLow
HSYNC active low.

enumerator kLCDIF_HsyncActiveHigh
HSYNC active high.

enumerator kLCDIF_DataEnableActiveLow
Data enable line active low.

enumerator kLCDIF_DataEnableActiveHigh
Data enable line active high.

enumerator kLCDIF_DriveDataOnFallingClkEdge
Drive data on falling clock edge, capture data on rising clock edge.

enumerator kLCDIF_DriveDataOnRisingClkEdge
Drive data on falling clock edge, capture data on rising clock edge.

enum _lcdif_output_format

LCDIF DPI output format.

Values:

enumerator kLCDIF_Output16BitConfig1
16-bit configuration 1. RGB565: XXXXXXXX_RRRRRGGG_GGGBBBBB.

enumerator kLCDIF_Output16BitConfig2
16-bit configuration 2. RGB565: XXXRRRRR_XXGGGGG_GGGBBBBB.

enumerator kLCDIF_Output16BitConfig3

16-bit configuration 3. RGB565: XXRRRRRX_XXGGGGGG_XXBBBBBX.

enumerator kLCDIF_Output18BitConfig1

18-bit configuration 1. RGB666: XXXXXRRR_RRRRGGGG_GGBBBBBB.

enumerator kLCDIF_Output18BitConfig2

18-bit configuration 2. RGB666: XXRRRRRR_XXGGGGGG_XXBBBBBB.

enumerator kLCDIF_Output24Bit

24-bit.

enum _lcdif_fb_format

LCDIF frame buffer pixel format.

Values:

enumerator kLCDIF_PixelFormatXRGB444

XRGB4444, deprecated, use kLCDIF_PixelFormatXRGB4444 instead.

enumerator kLCDIF_PixelFormatXRGB4444

XRGB4444, 16-bit each pixel, 4-bit each element. R4G4B4 in reference manual.

enumerator kLCDIF_PixelFormatXRGB1555

XRGB1555, 16-bit each pixel, 5-bit each element. R5G5B5 in reference manual.

enumerator kLCDIF_PixelFormatRGB565

RGB565, 16-bit each pixel. R5G6B5 in reference manual.

enumerator kLCDIF_PixelFormatXRGB8888

XRGB8888, 32-bit each pixel, 8-bit each element. R8G8B8 in reference manual.

enum _lcdif_interrupt

LCDIF interrupt and status.

Values:

enumerator kLCDIF_Display0FrameDoneInterrupt

The last pixel of visible area in frame is shown.

enum _lcdif_cursor_format

LCDIF cursor format.

Values:

enumerator kLCDIF_CursorMasked

Masked format.

enumerator kLCDIF_CursorARGB8888

ARGB8888.

enum _lcdif_dbi_cmd_flag

LCDIF DBI command flag.

Values:

enumerator kLCDIF_DbiCmdAddress

Send address (or command).

enumerator kLCDIF_DbiCmdWriteMem

Start write memory.

enumerator kLCDIF_DbiCmdData

Send data.

enumerator kLCDIF_DbiCmdReadMem

Start read memory.

enum _lcdif_dbi_out_format

LCDIF DBI output format.

Values:

enumerator kLCDIF_DbiOutD8RGB332

8-bit data bus width, pixel RGB332. For type A or B. 1 pixel sent in 1 cycle.

enumerator kLCDIF_DbiOutD8RGB444

8-bit data bus width, pixel RGB444. For type A or B. 2 pixels sent in 3 cycles.

enumerator kLCDIF_DbiOutD8RGB565

8-bit data bus width, pixel RGB565. For type A or B. 1 pixel sent in 2 cycles.

enumerator kLCDIF_DbiOutD8RGB666

8-bit data bus width, pixel RGB666. For type A or B. 1 pixel sent in 3 cycles, data bus 2 LSB not used.

enumerator kLCDIF_DbiOutD8RGB888

8-bit data bus width, pixel RGB888. For type A or B. 1 pixel sent in 3 cycles.

enumerator kLCDIF_DbiOutD9RGB666

9-bit data bus width, pixel RGB666. For type A or B. 1 pixel sent in 2 cycles.

enumerator kLCDIF_DbiOutD16RGB332

16-bit data bus width, pixel RGB332. For type A or B. 2 pixels sent in 1 cycle.

enumerator kLCDIF_DbiOutD16RGB444

16-bit data bus width, pixel RGB444. For type A or B. 1 pixel sent in 1 cycle, data bus 4 MSB not used.

enumerator kLCDIF_DbiOutD16RGB565

16-bit data bus width, pixel RGB565. For type A or B. 1 pixel sent in 1 cycle.

enumerator kLCDIF_DbiOutD16RGB666Option1

16-bit data bus width, pixel RGB666. For type A or B. 2 pixels sent in 3 cycles.

enumerator kLCDIF_DbiOutD16RGB666Option2

16-bit data bus width, pixel RGB666. For type A or B. 1 pixel sent in 2 cycles.

enumerator kLCDIF_DbiOutD16RGB888Option1

16-bit data bus width, pixel RGB888. For type A or B. 2 pixels sent in 3 cycles.

enumerator kLCDIF_DbiOutD16RGB888Option2

16-bit data bus width, pixel RGB888. For type A or B. 1 pixel sent in 2 cycles.

enumerator kLCDIF_DbiOutD1RGB565Option1

1-bit data bus width, pixel RGB565. For type C option 1, use extra bit to distinguish Data and Command (DC).

enumerator kLCDIF_DbiOutD1RGB565Option2

1-bit data bus width, pixel RGB565. For type C option 2, use extra byte to distinguish Data and Command (DC).

enumerator kLCDIF_DbiOutD1RGB565Option3

1-bit data bus width, pixel RGB565. For type C option 3, use extra DC line to distinguish Data and Command (DC).

enumerator kLCDIF_DbiOutD1RG888Option1

1-bit data bus width, pixel RGB888. For type C option 1, use extra bit to distinguish Data and Command (DC).

enumerator kLCDIF_DbiOutD1RG888Option2

1-bit data bus width, pixel RGB888. For type C option 2, use extra byte to distinguish Data and Command (DC).

enumerator kLCDIF_DbiOutD1RG888Option3

1-bit data bus width, pixel RGB888. For type C option 3, use extra DC line to distinguish Data and Command (DC).

enum _lcdif_dbi_type

LCDIF DBI type.

Values:

enumerator kLCDIF_DbiTypeA_FixedE

Selects DBI type A fixed E mode, 68000, Motorola mode.

enumerator kLCDIF_DbiTypeA_ClockedE

Selects DBI Type A Clocked E mode, 68000, Motorola mode.

enumerator kLCDIF_DbiTypeB

Selects DBI type B, 8080, Intel mode.

enumerator kLCDIF_DbiTypeC

Selects DBI type C, SPI mode.

enum _lcdif_dbi_out_swizzle

LCDIF DBI output swizzle.

Values:

enumerator kLCDIF_DbiOutSwizzleRGB

RGB

enumerator kLCDIF_DbiOutSwizzleBGR

BGR

typedef enum _lcdif_output_format lcdif_output_format_t

LCDIF DPI output format.

typedef struct _lcdif_dpi_config lcdif_dpi_config_t

Configuration for LCDIF module to work in DBI mode.

typedef enum _lcdif_fb_format lcdif_fb_format_t

LCDIF frame buffer pixel format.

typedef struct _lcdif_fb_config lcdif_fb_config_t

LCDIF frame buffer configuration.

typedef enum _lcdif_cursor_format lcdif_cursor_format_t

LCDIF cursor format.

typedef struct _lcdif_cursor_config lcdif_cursor_config_t

LCDIF cursor configuration.

typedef struct _lcdif_dither_config lcdif_dither_config_t

LCDIF dither configuration.

- a. Decide which bit of pixel color to enhance. This is configured by the `lcdif_dither_config_t::redSize`, `lcdif_dither_config_t::greenSize`, and `lcdif_dither_config_t::blueSize`. For example, setting `redSize=6` means it is the 6th bit starting from the MSB that we want to enhance, in other words, it is the `RedColor[2]bit` from `RedColor[7:0]`. `greenSize` and `blueSize` function in the same way.
- b. Create the look-up table.
 - a. The Look-Up Table includes 16 entries, 4 bits for each.
 - b. The Look-Up Table provides a value `U[3:0]` through the index `X[1:0]` and `Y[1:0]`.
 - c. The color value `RedColor[3:0]` is used to compare with this `U[3:0]`.
 - d. If `RedColor[3:0] > U[3:0]`, and `RedColor[7:2]` is not 6'b111111, then the final color value is: `NewRedColor = RedColor[7:2] + 1'b1`.
 - e. If `RedColor[3:0] <= U[3:0]`, then `NewRedColor = RedColor[7:2]`.

```
typedef enum _lcdif_dbi_out_format lcdif_dbi_out_format_t
```

LCDIF DBI output format.

```
typedef enum _lcdif_dbi_type lcdif_dbi_type_t
```

LCDIF DBI type.

```
typedef enum _lcdif_dbi_out_swizzle lcdif_dbi_out_swizzle_t
```

LCDIF DBI output swizzle.

```
typedef struct _lcdif_dbi_config lcdif_dbi_config_t
```

LCDIF DBI configuration.

```
LCDIF_MAKE_CURSOR_COLOR(r, g, b)
```

Construct the cursor color, every element should be in the range of 0 ~ 255.

```
LCDIF_MAKE_GAMMA_VALUE(r, g, b)
```

Construct the gamma value set to LCDIF gamma table, every element should be in the range of 0~255.

```
LCDIF_ALIGN_ADDR(addr, align)
```

Calculate the aligned address for LCDIF buffer.

```
LCDIF_FB_ALIGN
```

The frame buffer should be 128 byte aligned.

```
LCDIF_GAMMA_INDEX_MAX
```

Gamma index max value.

```
LCDIF_CURSOR_SIZE
```

The cursor size is 32 x 32.

```
LCDIF_FRAMEBUFFERCONFIG0_OUTPUT_MASK
```

```
LCDIF_ADDR_CPU_2_IP(addr)
```

```
struct _lcdif_dpi_config
```

`#include <fsl_lcdif.h>` Configuration for LCDIF module to work in DBI mode.

Public Members

```
uint16_t panelWidth
```

Display panel width, pixels per line.

```
uint16_t panelHeight
```

Display panel height, how many lines per panel.

```
uint8_t hsw
```

HSYNC pulse width.

uint8_t hfp

Horizontal front porch.

uint8_t hbp

Horizontal back porch.

uint8_t vsw

VSYNC pulse width.

uint8_t vfp

Vrtical front porch.

uint8_t vbp

Vertical back porch.

uint32_t polarityFlags

OR'ed value of _lcdif_polarity_flags, used to contol the signal polarity.

lcdif_output_format_t format

DPI output format.

struct _lcdif_fb_config

#include <fsl_lcdif.h> LCDIF frame buffer configuration.

Public Members

bool enable

Enable the frame buffer output.

bool enableGamma

Enable the gamma correction.

lcdif_fb_format_t format

Frame buffer pixel format.

struct _lcdif_cursor_config

#include <fsl_lcdif.h> LCDIF cursor configuration.

Public Members

bool enable

Enable the cursor or not.

lcdif_cursor_format_t format

Cursor format.

uint8_t hotspotOffsetX

Offset of the hotspot to top left point, range 0 ~ 31

uint8_t hotspotOffsetY

Offset of the hotspot to top left point, range 0 ~ 31

struct _lcdif_dither_config

#include <fsl_lcdif.h> LCDIF dither configuration.

- a. Decide which bit of pixel color to enhance. This is configured by the `lcdif_dither_config_t::redSize`, `lcdif_dither_config_t::greenSize`, and `lcdif_dither_config_t::blueSize`. For example, setting `redSize=6` means it is the 6th bit starting from the MSB that we want to enhance, in other words, it is the RedColor[2]bit from RedColor[7:0]. `greenSize` and `blueSize` function in the same way.

- b. Create the look-up table. a. The Look-Up Table includes 16 entries, 4 bits for each. b. The Look-Up Table provides a value U[3:0] through the index X[1:0] and Y[1:0]. c. The color value RedColor[3:0] is used to compare with this U[3:0]. d. If RedColor[3:0] > U[3:0], and RedColor[7:2] is not 6'b111111, then the final color value is: NewRedColor = RedColor[7:2] + 1'b1. e. If RedColor[3:0] <= U[3:0], then NewRedColor = RedColor[7:2].

Public Members

bool enable

Enable or not.

uint8_t redSize

Red color size, valid region 4-8.

uint8_t greenSize

Green color size, valid region 4-8.

uint8_t blueSize

Blue color size, valid region 4-8.

uint32_t low

Low part of the look up table.

uint32_t high

High part of the look up table.

struct _lcdif_dbi_config

#include <fsl_lcdif.h> LCDIF DBI configuration.

Public Members

lcdif_dbi_out_swizzle_t swizzle

Swizzle.

lcdif_dbi_out_format_t format

Output format.

uint8_t acTimeUnit

Time unit for AC characteristics.

lcdif_dbi_type_t type

DBI type.

bool reversePolarity

Reverse the DC pin polarity.

uint16_t writeWRPeriod

WR signal period, Cycle number = writeWRPeriod * (acTimeUnit + 1), must be no less than 3. Only for type A and type b.

uint8_t writeWRAssert

Cycle number = writeWRAssert * (acTimeUnit + 1), only for type A and type B. With kLCDIF_DbiTypeA_FixedE: Not used. With kLCDIF_DbiTypeA_ClockedE: Time to assert E. With kLCDIF_DbiTypeB: Time to assert WRX.

uint8_t writeCSAssert

Cycle number = writeCSAssert * (acTimeUnit + 1), only for type A and type B. With kLCDIF_DbiTypeA_FixedE: Time to assert CSX. With kLCDIF_DbiTypeA_ClockedE: Not used. With kLCDIF_DbiTypeB: Time to assert CSX.

uint16_t writeWRDeassert

Cycle number = writeWRDeassert * (acTimeUnit + 1), only for type A and type B. With kLCDIF_DbiTypeA_FixedE: Not used. With kLCDIF_DbiTypeA_ClockedE: Time to de-assert E. With kLCDIF_DbiTypeB: Time to de-assert WRX.

uint16_t writeCSDeassert

Cycle number = writeCSDeassert * (acTimeUnit + 1), only for type A and type B. With kLCDIF_DbiTypeA_FixedE: Time to de-assert CSX. With kLCDIF_DbiTypeA_ClockedE: Not used. With kLCDIF_DbiTypeB: Time to de-assert CSX.

uint8_t typeCTas

How many sdaClk cycles in Tas phase, only for Type C option 3, at least 1.

uint8_t typeCSCLTwrl

How many sdaClk cycles in Twrl phase, only for Type C, at least 1.

uint8_t typeCSCLTwrh

How many sdaClk cycles in Twrh phase, only for Type C, at least 1.

2.37 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 return the current event ID, LIN_RX_BUSY while receiving and return LIN_SUCCESS, or timeout (LIN_TIMEOUT) when the reception is complete. The bytesRemaining shows number of bytes that still needed to receive.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to receive

Returns

lin_status_t LIN_RX_BUSY, LIN_TIMEOUT or LIN_SUCCESS

lin_status_t LIN_LPUART_GoToSleepMode(LPUART_Type *base)

This function puts current node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_MODE.

Parameters

- base – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GotoIdleState(LPUART_Type *base)`

Puts current LIN node to Idle state This function changes current node state to `LIN_NODE_STATE_IDLE`.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_SendWakeupSignal(LPUART_Type *base)`

Sends a wakeup signal through the LIN_LPUART interface.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_MasterSendHeader(LPUART_Type *base, uint8_t id)`

Sends frame header out through the LIN_LPUART module using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- `base` – LPUART peripheral base address
- `id` – Frame Identifier

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_EnableIRQ(LPUART_Type *base)`

Enables LIN_LPUART hardware interrupts.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_DisableIRQ(LPUART_Type *base)`

Disables LIN_LPUART hardware interrupts.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_AutoBaudCapture(uint32_t instance)`

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- `instance` – LPUART instance

Returns

`lin_status_t`

```
void LIN_LPUART_IRQHandler(LPUART_Type *base)
```

LIN_LPUART RX TX interrupt handler.

Parameters

- base – LPUART peripheral base address

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
```

2.38 LPADC: 12-bit SAR Analog-to-Digital Converter Driver

enum `_lpadc_status_flags`

Define hardware flags of the module.

Values:

enumerator `kLPADC_ResultFIFO0OverflowFlag`

Indicates that more data has been written to the Result FIFO 0 than it can hold.

enumerator `kLPADC_ResultFIFO0ReadyFlag`

Indicates when the number of valid datawords in the result FIFO 0 is greater than the setting watermark level.

enumerator `kLPADC_TriggerExceptionFlag`

Indicates that a trigger exception event has occurred.

enumerator `kLPADC_TriggerCompletionFlag`

Indicates that a trigger completion event has occurred.

enumerator `kLPADC_CalibrationReadyFlag`

Indicates that the calibration process is done.

enumerator `kLPADC_ActiveFlag`

Indicates that the ADC is in active state.

enumerator `kLPADC_ResultFIFOOverflowFlag`

To compilitable with old version, do not recommend using this, please use `kLPADC_ResultFIFO0OverflowFlag` as instead.

enumerator `kLPADC_ResultFIFOReadyFlag`

To compilitable with old version, do not recommend using this, please use `kLPADC_ResultFIFO0ReadyFlag` as instead.

enum `_lpadc_interrupt_enable`

Define interrupt switchers of the module.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator `kLPADC_ResultFIFO0OverflowInterruptEnable`

Configures ADC to generate overflow interrupt requests when FOF0 flag is asserted.

enumerator `kLPADC_FIFO0WatermarkInterruptEnable`

Configures ADC to generate watermark interrupt requests when RDY0 flag is asserted.

enumerator `kLPADC_ResultFIFOOverflowInterruptEnable`

To compilitable with old version, do not recommend using this, please use `kLPADC_ResultFIFO0OverflowInterruptEnable` as instead.

enumerator `kLPADC_FIFOWatermarkInterruptEnable`

To compilitable with old version, do not recommend using this, please use `kLPADC_FIFO0WatermarkInterruptEnable` as instead.

enumerator `kLPADC_TriggerExceptionInterruptEnable`

Configures ADC to generate trigger exception interrupt.

enumerator `kLPADC_Trigger0CompletionInterruptEnable`

Configures ADC to generate interrupt when trigger 0 completion.

enumerator kLPADC_Trigger1CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 1 completion.

enumerator kLPADC_Trigger2CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 2 completion.

enumerator kLPADC_Trigger3CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 3 completion.

enumerator kLPADC_Trigger4CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 4 completion.

enumerator kLPADC_Trigger5CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 5 completion.

enumerator kLPADC_Trigger6CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 6 completion.

enumerator kLPADC_Trigger7CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 7 completion.

enumerator kLPADC_Trigger8CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 8 completion.

enumerator kLPADC_Trigger9CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 9 completion.

enumerator kLPADC_Trigger10CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 10 completion.

enumerator kLPADC_Trigger11CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 11 completion.

enumerator kLPADC_Trigger12CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 12 completion.

enumerator kLPADC_Trigger13CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 13 completion.

enumerator kLPADC_Trigger14CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 14 completion.

enumerator kLPADC_Trigger15CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 15 completion.

enum _lpadc_trigger_status_flags

The enumerator of lpadc trigger status flags, including interrupted flags and completed flags.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_Trigger0InterruptedFlag

Trigger 0 is interrupted by a high priority exception.

enumerator kLPADC_Trigger1InterruptedFlag

Trigger 1 is interrupted by a high priority exception.

enumerator kLPADC_Trigger2InterruptedFlag

Trigger 2 is interrupted by a high priority exception.

enumerator kLPADC_Trigger3InterruptedFlag

Trigger 3 is interrupted by a high priority exception.

enumerator kLPADC_Trigger4InterruptedFlag

Trigger 4 is interrupted by a high priority exception.

enumerator kLPADC_Trigger5InterruptedFlag

Trigger 5 is interrupted by a high priority exception.

enumerator kLPADC_Trigger6InterruptedFlag

Trigger 6 is interrupted by a high priority exception.

enumerator kLPADC_Trigger7InterruptedFlag

Trigger 7 is interrupted by a high priority exception.

enumerator kLPADC_Trigger8InterruptedFlag

Trigger 8 is interrupted by a high priority exception.

enumerator kLPADC_Trigger9InterruptedFlag

Trigger 9 is interrupted by a high priority exception.

enumerator kLPADC_Trigger10InterruptedFlag

Trigger 10 is interrupted by a high priority exception.

enumerator kLPADC_Trigger11InterruptedFlag

Trigger 11 is interrupted by a high priority exception.

enumerator kLPADC_Trigger12InterruptedFlag

Trigger 12 is interrupted by a high priority exception.

enumerator kLPADC_Trigger13InterruptedFlag

Trigger 13 is interrupted by a high priority exception.

enumerator kLPADC_Trigger14InterruptedFlag

Trigger 14 is interrupted by a high priority exception.

enumerator kLPADC_Trigger15InterruptedFlag

Trigger 15 is interrupted by a high priority exception.

enumerator kLPADC_Trigger0CompletedFlag

Trigger 0 is completed and trigger 0 has enabled completion interrupts.

enumerator kLPADC_Trigger1CompletedFlag

Trigger 1 is completed and trigger 1 has enabled completion interrupts.

enumerator kLPADC_Trigger2CompletedFlag

Trigger 2 is completed and trigger 2 has enabled completion interrupts.

enumerator kLPADC_Trigger3CompletedFlag

Trigger 3 is completed and trigger 3 has enabled completion interrupts.

enumerator kLPADC_Trigger4CompletedFlag

Trigger 4 is completed and trigger 4 has enabled completion interrupts.

enumerator kLPADC_Trigger5CompletedFlag

Trigger 5 is completed and trigger 5 has enabled completion interrupts.

enumerator kLPADC_Trigger6CompletedFlag

Trigger 6 is completed and trigger 6 has enabled completion interrupts.

enumerator kLPADC_Trigger7CompletedFlag

Trigger 7 is completed and trigger 7 has enabled completion interrupts.

enumerator kLPADC_Trigger8CompletedFlag

Trigger 8 is completed and trigger 8 has enabled completion interrupts.

enumerator kLPADC_Trigger9CompletedFlag

Trigger 9 is completed and trigger 9 has enabled completion interrupts.

enumerator kLPADC_Trigger10CompletedFlag

Trigger 10 is completed and trigger 10 has enabled completion interrupts.

enumerator kLPADC_Trigger11CompletedFlag

Trigger 11 is completed and trigger 11 has enabled completion interrupts.

enumerator kLPADC_Trigger12CompletedFlag

Trigger 12 is completed and trigger 12 has enabled completion interrupts.

enumerator kLPADC_Trigger13CompletedFlag

Trigger 13 is completed and trigger 13 has enabled completion interrupts.

enumerator kLPADC_Trigger14CompletedFlag

Trigger 14 is completed and trigger 14 has enabled completion interrupts.

enumerator kLPADC_Trigger15CompletedFlag

Trigger 15 is completed and trigger 15 has enabled completion interrupts.

enum _lpadc_sample_scale_mode

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Values:

enumerator kLPADC_SamplePartScale

Use divided input voltage signal. (For scale select, please refer to the reference manual).

enumerator kLPADC_SampleFullScale

Full scale (Factor of 1).

enum _lpadc_sample_channel_mode

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Values:

enumerator kLPADC_SampleChannelSingleEndSideA

Single-end mode, only A-side channel is converted.

enumerator kLPADC_SampleChannelSingleEndSideB

Single-end mode, only B-side channel is converted.

enumerator kLPADC_SampleChannelDiffBothSideAB

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDiffBothSideBA

Differential mode, the ADC result is (CHnB-CHnA).

enumerator kLPADC_SampleChannelDiffBothSide

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDualSingleEndBothSide

Dual-Single-Ended Mode. Both A side and B side channels are converted independently.

enum _lpadc_hardware_average_mode

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

Values:

enumerator kLPADC_HardwareAverageCount1

Single conversion.

enumerator kLPADC_HardwareAverageCount2

2 conversions averaged.

enumerator kLPADC_HardwareAverageCount4

4 conversions averaged.

enumerator kLPADC_HardwareAverageCount8

8 conversions averaged.

enumerator kLPADC_HardwareAverageCount16

16 conversions averaged.

enumerator kLPADC_HardwareAverageCount32

32 conversions averaged.

enumerator kLPADC_HardwareAverageCount64

64 conversions averaged.

enumerator kLPADC_HardwareAverageCount128

128 conversions averaged.

enum _lpadc_sample_time_mode

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Values:

enumerator kLPADC_SampleTimeADCK3

3 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK5

5 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK7

7 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK11

11 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK19
19 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK35
35 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK67
69 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK131
131 ADCK cycles total sample time.

enum _lpadc_hardware_compare_mode

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Values:

enumerator kLPADC_HardwareCompareDisabled
Compare disabled.

enumerator kLPADC_HardwareCompareStoreOnTrue
Compare enabled. Store on true.

enumerator kLPADC_HardwareCompareRepeatUntilTrue
Compare enabled. Repeat channel acquisition until true.

enum _lpadc_conversion_resolution_mode

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

Values:

enumerator kLPADC_ConversionResolutionStandard
Standard resolution. Single-ended 12-bit conversion, Differential 13-bit conversion with 2's complement output.

enumerator kLPADC_ConversionResolutionHigh
High resolution. Single-ended 16-bit conversion; Differential 16-bit conversion with 2's complement output.

enum _lpadc_conversion_average_mode

Define enumeration of conversion averages mode.

Configure the conversion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

Values:

enumerator kLPADC_ConversionAverage1
Single conversion.

enumerator kLPADC_ConversionAverage2
2 conversions averaged.

enumerator kLPADC_ConversionAverage4
4 conversions averaged.

enumerator kLPADC_ConversionAverage8
8 conversions averaged.

enumerator kLPADC_ConversionAverage16
16 conversions averaged.

enumerator kLPADC_ConversionAverage32
32 conversions averaged.

enumerator kLPADC_ConversionAverage64
64 conversions averaged.

enumerator kLPADC_ConversionAverage128
128 conversions averaged.

enum _lpadc_reference_voltage_mode

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

Values:

enumerator kLPADC_ReferenceVoltageAlt1
Option 1 setting.

enumerator kLPADC_ReferenceVoltageAlt2
Option 2 setting.

enumerator kLPADC_ReferenceVoltageAlt3
Option 3 setting.

enum _lpadc_power_level_mode

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Values:

enumerator kLPADC_PowerLevelAlt1
Lowest power setting.

enumerator kLPADC_PowerLevelAlt2
Next lowest power setting.

enumerator kLPADC_PowerLevelAlt3
...

enumerator kLPADC_PowerLevelAlt4
Highest power setting.

enum _lpadc_offset_calibration_mode

Define enumeration of offset calibration mode.

Values:

enumerator kLPADC_OffsetCalibration12bitMode
12 bit offset calibration mode.

enumerator kLPADC_OffsetCalibration16bitMode
16 bit offset calibration mode.

enum _lpadc_trigger_priority_policy

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

Values:

enumerator kLPADC_ConvPreemptImmediatelyNotAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion is not automatically resumed or restarted.

enumerator kLPADC_ConvPreemptSoftlyNotAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion is not resumed or restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoRestarted

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptSoftlyAutoRestarted

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be resumed.

enumerator kLPADC_ConvPreemptSoftlyAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will be automatically be resumed.

enumerator kLPADC_TriggerPriorityPreemptImmediately

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityPreemptSoftly

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityExceptionDisabled

High priority trigger exception disabled.

`typedef enum _lpadc_sample_scale_mode lpadc_sample_scale_mode_t`

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

`typedef enum _lpadc_sample_channel_mode lpadc_sample_channel_mode_t`

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

`typedef enum _lpadc_hardware_average_mode lpadc_hardware_average_mode_t`

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

`typedef enum _lpadc_sample_time_mode lpadc_sample_time_mode_t`

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

`typedef enum _lpadc_hardware_compare_mode lpadc_hardware_compare_mode_t`

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

`typedef enum _lpadc_conversion_resolution_mode lpadc_conversion_resolution_mode_t`

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to `lpadc_sample_channel_mode_t`

`typedef enum _lpadc_conversion_average_mode lpadc_conversion_average_mode_t`

Define enumeration of conversion averages mode.

Configure the conversion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

`typedef enum _lpadc_reference_voltage_mode lpadc_reference_voltage_source_t`

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

typedef enum *_lpadc_power_level_mode* lpadc_power_level_mode_t

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

typedef enum *_lpadc_offset_calibration_mode* lpadc_offset_calibration_mode_t

Define enumeration of offset calibration mode.

typedef enum *_lpadc_trigger_priority_policy* lpadc_trigger_priority_policy_t

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

typedef struct *_lpadc_calibration_value* lpadc_calibration_value_t

A structure of calibration value.

LPADC_CONVERSION_COMPLETE_TIMEOUT

Max loops to wait for LPADC conversion complete.

When doing calibration, driver will wait for the completion of conversion. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

LPADC_CALIBRATION_READY_TIMEOUT

Max loops to wait for LPADC calibration ready.

Before doing calibration, driver will wait for the calibration ready. This parameter defines how many loops to check the calibration ready. If defined as 0, driver will wait forever until ready.

LPADC_GAIN_CAL_READY_TIMEOUT

Max loops to wait for LPADC gain calibration GAIN_CAL ready.

Before doing calibration, driver will wait for the gain calibration GAIN_CAL ready. This parameter defines how many loops to check the gain calibration GAIN_CAL ready. If defined as 0, driver will wait forever until ready.

LPADC_GET_ACTIVE_COMMAND_STATUS(statusVal)

Define the MACRO function to get command status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

LPADC_GET_ACTIVE_TRIGGER_STATUE(statusVal)

Define the MACRO function to get trigger status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

void LPADC_Init(ADC_Type *base, const *lpadc_config_t* *config)

Initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.
- config – Pointer to configuration structure. See “*lpadc_config_t*”.

void LPADC_GetDefaultConfig(*lpadc_config_t* *config)

Gets an available pre-defined settings for initial configuration.

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

```
config->enableInDozeMode      = true;
config->enableAnalogPreliminary = false;
config->powerUpDelay           = 0x80;
config->referenceVoltageSource = kLPADC_ReferenceVoltageAlt1;
config->powerLevelMode         = kLPADC_PowerLevelAlt1;
config->triggerPriorityPolicy   = kLPADC_TriggerPriorityPreemptImmediately;
config->enableConvPause        = false;
config->convPauseDelay         = 0U;
config->FIFOWatermark          = 0U;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_Deinit(ADC_Type *base)

De-initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.

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

Switch on/off the LPADC module.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the module.

static inline void LPADC_DoResetFIFO(ADC_Type *base)

Do reset the conversion FIFO.

Parameters

- base – LPADC peripheral base address.

static inline void LPADC_DoResetConfig(ADC_Type *base)

Do reset the module's configuration.

Reset all ADC internal logic and registers, except the Control Register (ADCx_CTRL).

Parameters

- base – LPADC peripheral base address.

static inline uint32_t LPADC_GetStatusFlags(ADC_Type *base)

Get status flags.

Parameters

- base – LPADC peripheral base address.

Returns

status flags' mask. See to `_lpadc_status_flags`.

static inline void LPADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clear status flags.

Only the flags can be cleared by writing ADCx_STATUS register would be cleared by this API.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for flags to be cleared. See to `_lpadc_status_flags`.

static inline uint32_t LPADC_GetTriggerStatusFlags(ADC_Type *base)

Get trigger status flags to indicate which trigger sequences have been completed or interrupted by a high priority trigger exception.

Parameters

- base – LPADC peripheral base address.

Returns

The OR'ed value of `_lpadc_trigger_status_flags`.

static inline void LPADC_ClearTriggerStatusFlags(ADC_Type *base, uint32_t mask)

Clear trigger status flags.

Parameters

- base – LPADC peripheral base address.
- mask – The mask of trigger status flags to be cleared, should be the OR'ed value of `_lpadc_trigger_status_flags`.

static inline void LPADC_EnableInterrupts(ADC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

static inline void LPADC_DisableInterrupts(ADC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

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

Switch on/off the DMA trigger for FIFO watermark event.

Parameters

- base – LPADC peripheral base address.
- enable – Switcher to the event.

static inline uint32_t LPADC_GetConvResultCount(ADC_Type *base)

Get the count of result kept in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

Returns

The count of result kept in conversion FIFO.

bool LPADC_GetConvResult(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

- `result` – Pointer to structure variable that keeps the conversion result in conversion FIFO.

Returns

Status whether FIFO entry is valid.

`void LPADC_GetConvResultBlocking(ADC_Type *base, lpadc_conv_result_t *result)`

Get the result in conversion FIFO using blocking method.

Parameters

- `base` – LPADC peripheral base address.
- `result` – Pointer to structure variable that keeps the conversion result in conversion FIFO.

`void LPADC_SetConvTriggerConfig(ADC_Type *base, uint32_t triggerId, const lpadc_conv_trigger_config_t *config)`

Configure the conversion trigger source.

Each programmable trigger can launch the conversion command in command buffer.

Parameters

- `base` – LPADC peripheral base address.
- `triggerId` – ID for each trigger. Typically, the available value range is from 0.
- `config` – Pointer to configuration structure. See to `lpadc_conv_trigger_config_t`.

`void LPADC_GetDefaultConvTriggerConfig(lpadc_conv_trigger_config_t *config)`

Gets an available pre-defined settings for trigger's configuration.

This function initializes the trigger's configuration structure with an available settings. The default values are:

```
config->targetCommandId    = 0U;  
config->delayPower         = 0U;  
config->priority           = 0U;  
config->channelAFIFOSelect = 0U;  
config->channelBFIFOSelect = 0U;  
config->enableHardwareTrigger = false;
```

Parameters

- `config` – Pointer to configuration structure.

`static inline void LPADC_DoSoftwareTrigger(ADC_Type *base, uint32_t triggerIdMask)`

Do software trigger to conversion command.

Parameters

- `base` – LPADC peripheral base address.
- `triggerIdMask` – Mask value for software trigger indexes, which count from zero.

`void LPADC_SetConvCommandConfig(ADC_Type *base, uint32_t commandId, const lpadc_conv_command_config_t *config)`

Configure conversion command.

Note: The number of compare value register on different chips is different, that is mean in some chips, some command buffers do not have the compare functionality.

Parameters

- base – LPADC peripheral base address.
- commandId – ID for command in command buffer. Typically, the available value range is 1 - 15.
- config – Pointer to configuration structure. See to `lpadc_conv_command_config_t`.

void LPADC_GetDefaultConvCommandConfig(*lpadc_conv_command_config_t* *config)

Gets an available pre-defined settings for conversion command's configuration.

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
config->sampleScaleMode      = kLPADC_SampleFullScale;
config->channelBScaleMode    = kLPADC_SampleFullScale;
config->sampleChannelMode    = kLPADC_SampleChannelSingleEndSideA;
config->channelNumber        = 0U;
config->channelBNumber       = 0U;
config->chainedNextCommandNumber = 0U;
config->enableAutoChannelIncrement = false;
config->loopCount            = 0U;
config->hardwareAverageMode   = kLPADC_HardwareAverageCount1;
config->sampleTimeMode       = kLPADC_SampleTimeADCK3;
config->hardwareCompareMode   = kLPADC_HardwareCompareDisabled;
config->hardwareCompareValueHigh = 0U;
config->hardwareCompareValueLow  = 0U;
config->conversionResolutionMode = kLPADC_ConversionResolutionStandard;
config->enableWaitTrigger      = false;
config->enableChannelB        = false;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_EnableCalibration(ADC_Type *base, bool enable)

Enable the calibration function.

When CALOFS is set, the ADC is configured to perform a calibration function anytime the ADC executes a conversion. Any channel selected is ignored and the value returned in the RESFIFO is a signed value between -31 and 31. -32 is not a valid and is never a returned value. Software should copy the lower 6-bits of the conversion result stored in the RESFIFO after a completed calibration conversion to the OFSTRIM field. The OFSTRIM field is used in normal operation for offset correction.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetValue(ADC_Type *base, uint32_t value)

Set proper offset value to trim ADC.

To minimize the offset during normal operation, software should read the conversion result from the RESFIFO calibration operation and write the lower 6 bits to the OFSTRIM register.

Parameters

- base – LPADC peripheral base address.
- value – Setting offset value.

status_t LPADC_DoAutoCalibration(ADC_Type *base)

Do auto calibration.

Calibration function should be executed before using converter in application. It used the software trigger and a dummy conversion, get the offset and write them into the OFSTRIM register. It called some of functional API including: -LPADC_EnableCalibration(...) -LPADC_LPADC_SetOffsetValue(...) -LPADC_SetConvCommandConfig(...) -LPADC_SetConvTriggerConfig(...)

Parameters

- base – LPADC peripheral base address.
- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

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

Enable the offset calibration function.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetCalibrationMode(ADC_Type *base,
lpadc_offset_calibration_mode_t mode)

Set offset calibration mode.

Parameters

- base – LPADC peripheral base address.
- mode – set offset calibration mode.see to *lpadc_offset_calibration_mode_t*.

status_t LPADC_DoOffsetCalibration(ADC_Type *base)

Do offset calibration.

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_PrepareAutoCalibration(ADC_Type *base)

Prepare auto calibration, LPADC_FinishAutoCalibration has to be called before using the LPADC. LPADC_DoAutoCalibration has been split in two API to avoid to be stuck too long in the function.

Parameters

- base – LPADC peripheral base address.

status_t LPADC_FinishAutoCalibration(ADC_Type *base)

Finish auto calibration start with LPADC_PrepareAutoCalibration.

Note: This feature is used for LPADC with CTRL[CALOFSMODE].

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
void LPADC_GetCalibrationValue(ADC_Type *base, lpadc_calibration_value_t  
                             *ptrCalibrationValue)
```

Get calibration value into the memory which is defined by invoker.

Note: Please note the ADC will be disabled temporary.

Note: This function should be used after finish calibration.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure, this memory block should be always powered on even in low power modes.

```
status_t LPADC_SetCalibrationValue(ADC_Type *base, const lpadc_calibration_value_t  
                                  *ptrCalibrationValue)
```

Set calibration value into ADC calibration registers.

Note: Please note the ADC will be disabled temporary.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure which contains ADC's calibration value.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

FSL_LPADC_DRIVER_VERSION

LPADC driver version 2.9.3.

```
struct lpadc_config_t
```

```
#include <fsl_lpadc.h> LPADC global configuration.
```

This structure would used to keep the settings for initialization.

Public Members

```
bool enableInternalClock
```

Enables the internally generated clock source. The clock source is used in clock selection logic at the chip level and is optionally used for the ADC clock source.

```
bool enableVref1LowVoltage
```

If voltage reference option1 input is below 1.8V, it should be “true”. If voltage reference option1 input is above 1.8V, it should be “false”.

`bool enableInDozeMode`

Control system transition to Stop and Wait power modes while ADC is converting. When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

`lpadc_conversion_average_mode_t` conversionAverageMode

Auto-Calibration Averages.

`bool enableAnalogPreliminary`

ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).

`uint32_t powerUpDelay`

When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize. The startup delay count of (powerUpDelay * 4) ADCK cycles must result in a longer delay than the analog startup time.

`lpadc_reference_voltage_source_t` referenceVoltageSource

Selects the voltage reference high used for conversions.

`lpadc_power_level_mode_t` powerLevelMode

Power Configuration Selection.

`lpadc_trigger_priority_policy_t` triggerPriorityPolicy

Control how higher priority triggers are handled, see to `lpadc_trigger_priority_t`.

`bool enableConvPause`

Enables the ADC pausing function. When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in “Compare Until True” configuration.

`uint32_t convPauseDelay`

Controls the duration of pausing during command execution sequencing. The pause delay is a count of (convPauseDelay*4) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

`uint32_t FIFOWatermark`

FIFOWatermark is a programmable threshold setting. When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

`struct lpadc_conv_command_config_t`

`#include <fsl_lpadc.h>` Define structure to keep the configuration for conversion command.

Public Members

`lpadc_sample_scale_mode_t` sampleScaleMode

Sample scale mode.

`lpadc_sample_scale_mode_t` channelBScaleMode

Alternate channel B Scale mode.

`lpadc_sample_channel_mode_t` sampleChannelMode

Channel sample mode.

uint32_t channelNumber

Channel number, select the channel or channel pair.

uint32_t channelBNumber

Alternate Channel B number, select the channel.

uint32_t chainedNextCommandNumber

Selects the next command to be executed after this command completes. 1-15 is available, 0 is to terminate the chain after this command.

bool enableAutoChannelIncrement

Loop with increment: when disabled, the “loopCount” field selects the number of times the selected channel is converted consecutively; when enabled, the “loopCount” field defines how many consecutive channels are converted as part of the command execution.

uint32_t loopCount

Selects how many times this command executes before finish and transition to the next command or Idle state. Command executes LOOP+1 times. 0-15 is available.

lpadc_hardware_average_mode_t hardwareAverageMode

Hardware average selection.

lpadc_sample_time_mode_t sampleTimeMode

Sample time selection.

lpadc_hardware_compare_mode_t hardwareCompareMode

Hardware compare selection.

uint32_t hardwareCompareValueHigh

Compare Value High. The available value range is in 16-bit.

uint32_t hardwareCompareValueLow

Compare Value Low. The available value range is in 16-bit.

lpadc_conversion_resolution_mode_t conversionResolutionMode

Conversion resolution mode.

bool enableWaitTrigger

Wait for trigger assertion before execution: when disabled, this command will be automatically executed; when enabled, the active trigger must be asserted again before executing this command.

struct *lpadc_conv_trigger_config_t*

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion trigger.

Public Members

uint32_t targetCommandId

Select the command from command buffer to execute upon detect of the associated trigger event.

uint32_t delayPower

Select the trigger delay duration to wait at the start of servicing a trigger event. When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is $2^{\text{delayPower}}$ ADCK cycles. The available value range is 4-bit.

uint32_t priority

Sets the priority of the associated trigger source. If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

bool enableHardwareTrigger

Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not. The software trigger is always available.

struct lpadc_conv_result_t

#include <fsl_lpadc.h> Define the structure to keep the conversion result.

Public Members

uint32_t commandIdSource

Indicate the command buffer being executed that generated this result.

uint32_t loopCountIndex

Indicate the loop count value during command execution that generated this result.

uint32_t triggerIdSource

Indicate the trigger source that initiated a conversion and generated this result.

uint16_t convValue

Data result.

struct __lpadc_calibration_value

#include <fsl_lpadc.h> A structure of calibration value.

2.39 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.

2.40 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;
```

(continues on next page)

(continued from previous page)

```

masterConfig->sciGlitchFilterWidth_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.

2.41 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.

2.42 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;
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 enableRxDatSCLStall 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.

2.43 LPIT: Low-Power Interrupt Timer

enum _lpit_chnl
List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

Values:

enumerator kLPIT_Chnl_0
LPIT channel number 0

enumerator kLPIT_Chnl_1
LPIT channel number 1

enumerator kLPIT_Chnl_2
LPIT channel number 2

enumerator kLPIT_Chnl_3
LPIT channel number 3

enum _lpit_timer_modes
Mode options available for the LPIT timer.

Values:

enumerator kLPIT_PeriodicCounter
Use the all 32-bits, counter loads and decrements to zero

enumerator kLPIT_DualPeriodicCounter
Counter loads, lower 16-bits decrement to zero, then upper 16-bits decrement

enumerator kLPIT_TriggerAccumulator
Counter loads on first trigger and decrements on each trigger

enumerator kLPIT_InputCapture
Counter loads with 0xFFFFFFFF, decrements to zero. It stores the inverse of the current value when a input trigger is detected

enum _lpit_trigger_select
Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

Values:

enumerator kLPIT_Trigger_TimerChn0
Channel 0 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn1
Channel 1 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn2
Channel 2 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn3
Channel 3 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn4
Channel 4 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn5
Channel 5 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn6
Channel 6 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn7
Channel 7 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn8
Channel 8 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn9
Channel 9 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn10

Channel 10 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn11

Channel 11 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn12

Channel 12 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn13

Channel 13 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn14

Channel 14 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn15

Channel 15 is selected as a trigger source

enum _lpit_trigger_source

Trigger source options available.

Values:

enumerator kLPIT_TriggerSource_External

Use external trigger input

enumerator kLPIT_TriggerSource_Internal

Use internal trigger

enum _lpit_interrupt_enable

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerInterruptEnable

Channel 0 Timer interrupt

enumerator kLPIT_Channel1TimerInterruptEnable

Channel 1 Timer interrupt

enumerator kLPIT_Channel2TimerInterruptEnable

Channel 2 Timer interrupt

enumerator kLPIT_Channel3TimerInterruptEnable

Channel 3 Timer interrupt

enum _lpit_status_flags

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerFlag

Channel 0 Timer interrupt flag

enumerator kLPIT_Channel1TimerFlag

Channel 1 Timer interrupt flag

enumerator kLPIT_Channel2TimerFlag

Channel 2 Timer interrupt flag

enumerator kLPIT_Channel3TimerFlag

Channel 3 Timer interrupt flag

typedef enum *_lpit_chnl* lpit_chnl_t

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

typedef enum *_lpit_timer_modes* lpit_timer_modes_t

Mode options available for the LPIT timer.

typedef enum *_lpit_trigger_select* lpit_trigger_select_t

Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

typedef enum *_lpit_trigger_source* lpit_trigger_source_t

Trigger source options available.

typedef enum *_lpit_interrupt_enable* lpit_interrupt_enable_t

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef enum *_lpit_status_flags* lpit_status_flags_t

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef struct *_lpit_chnl_params* lpit_chnl_params_t

Structure to configure the channel timer.

typedef struct *_lpit_config* lpit_config_t

LPIT configuration structure.

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

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

FSL_LPIT_DRIVER_VERSION

Version 2.1.2

LPIT_RESET_STATE_DELAY

Delay used in LPIT_Reset.

The macro value should be larger than $4 * \text{core clock} / \text{LPIT peripheral clock}$.

void LPIT_Init(LPIT_Type *base, const *lpit_config_t* *config)

Ungates the LPIT clock and configures the peripheral for a basic operation.

This function issues a software reset to reset all channels and registers except the Module Control register.

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

Parameters

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

void LPIT_Deinit(LPIT_Type *base)

Disables the module and gates the LPIT clock.

Parameters

- base – LPIT peripheral base address.

void LPIT_GetDefaultConfig(*lpit_config_t* *config)

Fills in the LPIT configuration structure with default settings.

The default values are:

```
config->enableRunInDebug = false;  
config->enableRunInDoze = false;
```

Parameters

- config – Pointer to the user configuration structure.

status_t LPIT_SetupChannel(LPIT_Type *base, *lpit_chnl_t* channel, const *lpit_chnl_params_t* *chnlSetup)

Sets up an LPIT channel based on the user's preference.

This function sets up the operation mode to one of the options available in the enumeration *lpit_timer_modes_t*. It sets the trigger source as either internal or external, trigger selection and the timers behaviour when a timeout occurs. It also chains the timer if a prior timer if requested by the user.

Parameters

- base – LPIT peripheral base address.
- channel – Channel that is being configured.
- chnlSetup – Configuration parameters.

static inline void LPIT_EnableInterrupts(LPIT_Type *base, uint32_t mask)

Enables the selected PIT interrupts.

Parameters

- base – LPIT peripheral base address.
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *lpit_interrupt_enable_t*

static inline void LPIT_DisableInterrupts(LPIT_Type *base, uint32_t mask)

Disables the selected PIT interrupts.

Parameters

- base – LPIT peripheral base address.
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *lpit_interrupt_enable_t*

```
static inline uint32_t LPIT_GetEnabledInterrupts(LPIT_Type *base)
```

Gets the enabled LPIT interrupts.

Parameters

- base – LPIT peripheral base address.

Returns

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

```
static inline uint32_t LPIT_GetStatusFlags(LPIT_Type *base)
```

Gets the LPIT status flags.

Parameters

- base – LPIT peripheral base address.

Returns

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

```
static inline void LPIT_ClearStatusFlags(LPIT_Type *base, uint32_t mask)
```

Clears the LPIT status flags.

Parameters

- base – LPIT peripheral base address.
- mask – The status flags to clear. This is a logical OR of members of the enumeration `lpit_status_flags_t`

```
static inline void LPIT_SetTimerPeriod(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

Timers begin counting down from the value set by this function until it reaches 0, at which point it generates an interrupt and loads this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

```
static inline void LPIT_SetTimerValue(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

In the Dual 16-bit Periodic Counter mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.

Note: Set TVAL register to 0 or 1 is invalid in compare mode.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

- ticks – Timer period in units of ticks.

static inline uint32_t LPIT_GetCurrentTimerCount(LPIT_Type *base, *lpit_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to microseconds or milliseconds.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

Returns

Current timer counting value in ticks.

static inline void LPIT_StartTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Starts the timer counting.

After calling this function, timers load the period value and count down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

static inline void LPIT_StopTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Stops the timer counting.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

static void LPIT_ResetStateDelay(void)

Short wait for LPIT state reset.

After clear or set LPIT_EN, there should be delay longer than 4 LPIT functional clock.

Parameters

- count – Delay count.

static inline void LPIT_Reset(LPIT_Type *base)

Performs a software reset on the LPIT module.

This resets all channels and registers except the Module Control Register.

Parameters

- base – LPIT peripheral base address.

struct *__lpit_chnl_params*

#include <fsl_lpit.h> Structure to configure the channel timer.

Public Members

bool chainChannel

true: Timer chained to previous timer; false: Timer not chained

lpit_timer_modes_t timerMode

Timers mode of operation.

lpit_trigger_select_t triggerSelect

Trigger selection for the timer

lpit_trigger_source_t triggerSource

Decides if we use external or internal trigger.

bool enableReloadOnTrigger

true: Timer reloads when a trigger is detected; false: No effect

bool enableStopOnTimeout

true: Timer will stop after timeout; false: does not stop after timeout

bool enableStartOnTrigger

true: Timer starts when a trigger is detected; false: decrement immediately

struct *_lpit_config*

#include <fsl_lpit.h> LPIT configuration structure.

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

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

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

bool enableRunInDoze

true: Timers run in doze mode; false: Timers stop in doze mode

2.44 LPSPI: Low Power Serial Peripheral Interface

2.45 LPSPI Peripheral driver

void LPSPI_MasterInit(LPSPI_Type *base, const *lpspi_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Initializes the LPSPI master.

Parameters

- base – LPSPI peripheral address.
- masterConfig – Pointer to structure *lpspi_master_config_t*.
- srcClock_Hz – Module source input clock in Hertz

void LPSPI_MasterGetDefaultConfig(*lpspi_master_config_t* *masterConfig)

Sets the *lpspi_master_config_t* structure to default values.

This API initializes the configuration structure for LPSPI_MasterInit(). The initialized structure can remain unchanged in LPSPI_MasterInit(), or can be modified before calling the LPSPI_MasterInit(). Example:

```
lpspi_master_config_t masterConfig;  
LPSPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – pointer to lpspi_master_config_t structure

void LPSPI_SlaveInit(LPSPI_Type *base, const *lpspi_slave_config_t* *slaveConfig)
LPSPI slave configuration.

Parameters

- base – LPSPI peripheral address.
- slaveConfig – Pointer to a structure lpspi_slave_config_t.

void LPSPI_SlaveGetDefaultConfig(*lpspi_slave_config_t* *slaveConfig)
Sets the lpspi_slave_config_t structure to default values.

This API initializes the configuration structure for LPSPI_SlaveInit(). The initialized structure can remain unchanged in LPSPI_SlaveInit() or can be modified before calling the LPSPI_SlaveInit(). Example:

```
lpspi_slave_config_t slaveConfig;  
LPSPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – pointer to lpspi_slave_config_t structure.

void LPSPI_Deinit(LPSPI_Type *base)
De-initializes the LPSPI peripheral. Call this API to disable the LPSPI clock.

Parameters

- base – LPSPI peripheral address.

void LPSPI_Reset(LPSPI_Type *base)
Restores the LPSPI peripheral to reset state. Note that this function sets all registers to reset state. As a result, the LPSPI module can't work after calling this API.

Parameters

- base – LPSPI peripheral address.

uint32_t LPSPI_GetInstance(LPSPI_Type *base)
Get the LPSPI instance from peripheral base address.

Parameters

- base – LPSPI peripheral base address.

Returns

LPSPI instance.

static inline void LPSPI_Enable(LPSPI_Type *base, bool enable)
Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Parameters

- base – LPSPI peripheral address.
- enable – Pass true to enable module, false to disable module.

```
static inline uint32_t LPSPI_GetStatusFlags(LPSPI_Type *base)
```

Gets the LPSPI status flag state.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI status(in SR register).

```
static inline uint8_t LPSPI_GetTxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Tx FIFO size.

```
static inline uint8_t LPSPI_GetRxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Rx FIFO size.

```
static inline uint32_t LPSPI_GetTxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the transmit FIFO.

```
static inline uint32_t LPSPI_GetRxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the receive FIFO.

```
static inline void LPSPI_ClearStatusFlags(LPSPI_Type *base, uint32_t statusFlags)
```

Clears the LPSPI status flag.

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the `_lpspi_flags`. Example usage:

```
LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag|kLPSPI_RxDataReadyFlag);
```

Parameters

- base – LPSPI peripheral address.
- statusFlags – The status flag used from type `_lpspi_flags`.

```
static inline uint32_t LPSPI_GetTcr(LPSPI_Type *base)
```

```
static inline void LPSPI_EnableInterrupts(LPSPI_Type *base, uint32_t mask)
```

Enables the LPSPI interrupts.

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

```
static inline void LPSPI_DisableInterrupts(LPSPI_Type *base, uint32_t mask)
```

Disables the LPSPI interrupts.

```
LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

```
static inline void LPSPI_EnableDMA(LPSPI_Type *base, uint32_t mask)
```

Enables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

```
static inline void LPSPI_DisableDMA(LPSPI_Type *base, uint32_t mask)
```

Disables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
SPI_DisableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

```
static inline uint32_t LPSPI_GetTxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Transmit Data Register address for a DMA operation.

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Transmit Data Register address.


```
static inline uint32_t LPSPI_GetRxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Receive Data Register address for a DMA operation.

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Receive Data Register address.

```
bool LPSPI_CheckTransferArgument(LPSPI_Type *base, lpspi_transfer_t *transfer, bool isEdma)
```

Check the argument for transfer .

Parameters

- base – LPSPI peripheral address.
- transfer – the transfer struct to be used.
- isEdma – True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns

Return true for right and false for wrong.

```
static inline void LPSPI_SetMasterSlaveMode(LPSPI_Type *base, lpspi_master_slave_mode_t mode)
```

Configures the LPSPI for either master or slave.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

- base – LPSPI peripheral address.
- mode – Mode setting (master or slave) of type lpspi_master_slave_mode_t.

```
static inline void LPSPI_SelectTransferPCS(LPSPI_Type *base, lpspi_which_pcs_t select)
```

Configures the peripheral chip select used for the transfer.

Parameters

- base – LPSPI peripheral address.
- select – LPSPI Peripheral Chip Select (PCS) configuration.

```
static inline void LPSPI_SetPCSContinuous(LPSPI_Type *base, bool IsContinuous)
```

Set the PCS signal to continuous or uncontinuous mode.

Note: In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

- base – LPSPI peripheral address.
- IsContinuous – True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

```
static inline bool LPSPI_IsMaster(LPSPI_Type *base)
```

Returns whether the LPSPI module is in master mode.

Parameters

- base – LPSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

```
static inline void LPSPI_FlushFifo(LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)
```

Flushes the LPSPI FIFOs.

Parameters

- base – LPSPI peripheral address.
- flushTxFifo – Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
- flushRxFifo – Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

```
static inline void LPSPI_SetFifoWatermarks(LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)
```

Sets the transmit and receive FIFO watermark values.

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

- base – LPSPI peripheral address.
- txWater – The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
- rxWater – The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

```
static inline void LPSPI_SetAllPcsPolarity(LPSPI_Type *base, uint32_t mask)
```

Configures all LPSPI peripheral chip select polarities simultaneously.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow | kLPSPI_Pcs1ActiveLow);
```

Parameters

- base – LPSPI peripheral address.
- mask – The PCS polarity mask; Use the enum `_lpspi_pcs_polarity`.

```
static inline void LPSPI_SetFrameSize(LPSPI_Type *base, uint32_t frameSize)
```

Configures the frame size.

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPI in slave mode, although the command register does not update until after the LPSPI is enabled. After it is enabled, the transmit command register should only be changed if the LPSPI is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

- base – LPSPI peripheral address.
- frameSize – The frame size in number of bits.

```
uint32_t LPSPI_MasterSetBaudRate(LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t
                                srcClock_Hz, uint32_t *tcrPrescaleValue)
```

Sets the LPSPI baud rate in bits per second.

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale tcrPrescaleValue parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- baudRate_Bps – The desired baud rate in bits per second.
- srcClock_Hz – Module source input clock in Hertz.
- tcrPrescaleValue – The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a “0” if the LPSPI is not configured for master mode or if the LPSPI module is not disabled.

```
void LPSPI_MasterSetDelayScaler(LPSPI_Type *base, uint32_t scaler, lpspi_delay_type_t
                                whichDelay)
```

Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- scaler – The 8-bit delay value 0x00 to 0xFF (255).

- `whichDelay` – The desired delay to configure, must be of type `lpspi_delay_type_t`.

`uint32_t LPSPI_MasterSetDelayTimes(LPSPI_Type *base, uint32_t delayTimeInNanoSec, lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)`

Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPI module must be configured for master mode before configuring this. And note that the `delayTime = LPSPI_clockSource / (PRESCALE * Delay_scaler)`.

Parameters

- `base` – LPSPI peripheral address.
- `delayTimeInNanoSec` – The desired delay value in nano-seconds.
- `whichDelay` – The desired delay to configuration, which must be of type `lpspi_delay_type_t`.
- `srcClock_Hz` – Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

`static inline void LPSPI_WriteData(LPSPI_Type *base, uint32_t data)`

Writes data into the transmit data buffer.

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

- `base` – LPSPI peripheral address.
- `data` – The data word to be sent.

`static inline uint32_t LPSPI_ReadData(LPSPI_Type *base)`

Reads data from the data buffer.

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

- `base` – LPSPI peripheral address.

Returns

The data read from the data buffer.

```
void LPSPI_SetDummyData(LPSPI_Type *base, uint8_t dummyData)
```

Set up the dummy data.

Parameters

- *base* – LPSPI peripheral address.
- *dummyData* – Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

```
void LPSPI_MasterTransferCreateHandle(LPSPI_Type *base, lpspi_master_handle_t *handle,
                                     lpspi_master_transfer_callback_t callback, void
                                     *userData)
```

Initializes the LPSPI master handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – LPSPI handle pointer to *lpspi_master_handle_t*.
- *callback* – DSPI callback.
- *userData* – callback function parameter.

```
status_t LPSPI_MasterTransferBlocking(LPSPI_Type *base, lpspi_transfer_t *transfer)
```

LPSPI master transfer data using a polling method.

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *transfer* – pointer to *lpspi_transfer_t* structure.

Returns

status of status_t.

```
status_t LPSPI_MasterTransferNonBlocking(LPSPI_Type *base, lpspi_master_handle_t *handle,
                                         lpspi_transfer_t *transfer)
```

LPSPI master transfer data using an interrupt method.

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – pointer to *lpspi_master_handle_t* structure which stores the transfer state.

- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_MasterTransferGetCount(LPSPI_Type *base, *lpspi_master_handle_t* *handle, size_t *count)

Gets the master transfer remaining bytes.

This function gets the master transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

`void` LPSPI_MasterTransferAbort(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI master abort transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

`void` LPSPI_MasterTransferHandleIRQ(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI Master IRQ handler function.

This function processes the LPSPI transmit and receive IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

`void` LPSPI_SlaveTransferCreateHandle(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_slave_transfer_callback_t* callback, void *userData)

Initializes the LPSPI slave handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to `lpspi_slave_handle_t`.
- callback – DSPI callback.
- userData – callback function parameter.

`status_t` LPSPI_SlaveTransferNonBlocking(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfer data using an interrupt method.

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_SlaveTransferGetCount(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, size_t *count)

Gets the slave transfer remaining bytes.

This function gets the slave transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

void LPSPI_SlaveTransferAbort(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI slave aborts a transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

void LPSPI_SlaveTransferHandleIRQ(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI Slave IRQ handler function.

This function processes the LPSPI transmit and receives an IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

bool LPSPI_WaitTxFifoEmpty(LPSPI_Type *base)

Wait for tx FIFO to be empty.

This function wait the tx fifo empty

Parameters

- base – LPSPI peripheral address.

Returns

true for the tx FIFO is ready, false is not.

void LPSPI_DriverIRQHandler(uint32_t instance)

LPSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for LPSPI.

Parameters

- instance – LPSPI instance.

FSL_LPSPI_DRIVER_VERSION

LPSPI driver version.

Status for the LPSPI driver.

Values:

enumerator kStatus_LPSPI_Busy

LPSPI transfer is busy.

enumerator kStatus_LPSPI_Error

LPSPI driver error.

enumerator kStatus_LPSPI_Idle

LPSPI is idle.

enumerator kStatus_LPSPI_OutOfRange

LPSPI transfer out Of range.

enumerator kStatus_LPSPI_Timeout

LPSPI timeout polling status flags.

enum _lpspi_flags

LPSPI status flags in SPIx_SR register.

Values:

enumerator kLPSPI_TxDataRequestFlag

Transmit data flag

enumerator kLPSPI_RxDataReadyFlag

Receive data flag

enumerator kLPSPI_WordCompleteFlag

Word Complete flag

enumerator kLPSPI_FrameCompleteFlag

Frame Complete flag

enumerator kLPSPI_TransferCompleteFlag

Transfer Complete flag

enumerator kLPSPI_TransmitErrorFlag

Transmit Error flag (FIFO underrun)

enumerator kLPSPI_ReceiveErrorFlag

Receive Error flag (FIFO overrun)

enumerator kLPSPI_DataMatchFlag

Data Match flag

enumerator kLPSPI_ModuleBusyFlag

Module Busy flag

enumerator kLPSPI_AllStatusFlag

Used for clearing all w1c status flags

enum _lpspi_interrupt_enable

LPSPI interrupt source.

Values:

enumerator kLPSPI_TxInterruptEnable

Transmit data interrupt enable

enumerator kLPSPI_RxInterruptEnable

Receive data interrupt enable

enumerator kLPSPI_WordCompleteInterruptEnable

Word complete interrupt enable

enumerator kLPSPI_FrameCompleteInterruptEnable

Frame complete interrupt enable

enumerator kLPSPI_TransferCompleteInterruptEnable

Transfer complete interrupt enable

enumerator kLPSPI_TransmitErrorInterruptEnable

Transmit error interrupt enable(FIFO underrun)

enumerator kLPSPI_ReceiveErrorInterruptEnable

Receive Error interrupt enable (FIFO overrun)

enumerator kLPSPI_DataMatchInterruptEnable

Data Match interrupt enable

enumerator kLPSPI_AllInterruptEnable

All above interrupts enable.

enum _lpspi_dma_enable

LPSPI DMA source.

Values:

enumerator kLPSPI_TxDmaEnable

Transmit data DMA enable

enumerator kLPSPI_RxDmaEnable

Receive data DMA enable

enum _lpspi_master_slave_mode

LPSPI master or slave mode configuration.

Values:

enumerator kLPSPI_Master

LPSPI peripheral operates in master mode.

enumerator kLPSPI_Slave

LPSPI peripheral operates in slave mode.

enum _lpspi_which_pcs_config

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

Values:

enumerator kLPSPI_Pcs0

PCS[0]

enumerator kLPSPI_Pcs1
PCS[1]

enumerator kLPSPI_Pcs2
PCS[2]

enumerator kLPSPI_Pcs3
PCS[3]

enum __lpspi_pcs_polarity_config
LPSPI Peripheral Chip Select (PCS) Polarity configuration.
Values:

enumerator kLPSPI_PcsActiveHigh
PCS Active High (idles low)

enumerator kLPSPI_PcsActiveLow
PCS Active Low (idles high)

enum __lpspi_pcs_polarity
LPSPI Peripheral Chip Select (PCS) Polarity.
Values:

enumerator kLPSPI_Pcs0ActiveLow
Pcs0 Active Low (idles high).

enumerator kLPSPI_Pcs1ActiveLow
Pcs1 Active Low (idles high).

enumerator kLPSPI_Pcs2ActiveLow
Pcs2 Active Low (idles high).

enumerator kLPSPI_Pcs3ActiveLow
Pcs3 Active Low (idles high).

enumerator kLPSPI_PcsAllActiveLow
Pcs0 to Pcs5 Active Low (idles high).

enum __lpspi_clock_polarity
LPSPI clock polarity configuration.
Values:

enumerator kLPSPI_ClockPolarityActiveHigh
CPOL=0. Active-high LPSPI clock (idles low)

enumerator kLPSPI_ClockPolarityActiveLow
CPOL=1. Active-low LPSPI clock (idles high)

enum __lpspi_clock_phase
LPSPI clock phase configuration.
Values:

enumerator kLPSPI_ClockPhaseFirstEdge
CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

enumerator kLPSPI_ClockPhaseSecondEdge
CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

enum `_lpspi_shift_direction`

LPSPi data shifter direction options.

Values:

enumerator `kLPSPI_MsbFirst`

Data transfers start with most significant bit.

enumerator `kLPSPI_LsbFirst`

Data transfers start with least significant bit.

enum `_lpspi_host_request_select`

LPSPi Host Request select configuration.

Values:

enumerator `kLPSPI_HostReqExtPin`

Host Request is an ext pin.

enumerator `kLPSPI_HostReqInternalTrigger`

Host Request is an internal trigger.

enum `_lpspi_match_config`

LPSPi Match configuration options.

Values:

enumerator `kLPSI_MatchDisabled`

LPSPi Match Disabled.

enumerator `kLPSI_1stWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSI_AnyWordEqualsM0orM1`

LPSPi Match Enabled.

enumerator `kLPSI_1stWordEqualsM0and2ndWordEqualsM1`

LPSPi Match Enabled.

enumerator `kLPSI_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` you flag is used or not, the waveform is 1 2 3 4 5 6 7 8.

- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_MasterByteSwap flag.

enum _lpspi_transfer_config_flag_for_slave

Use this enumeration for LPSPi slave transfer configFlags.

Values:

enumerator kLPSPi_SlavePcs0

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS0 signal

enumerator kLPSPi_SlavePcs1

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS1 signal

enumerator kLPSPi_SlavePcs2

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS2 signal

enumerator kLPSPi_SlavePcs3

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS3 signal

enumerator kLPSPi_SlaveByteSwap

Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

- i. If you set bitPerFrame = 8 , no matter the kLPSPi_SlaveByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.
- iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPi_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPi_SlaveByteSwap flag.

enum _lpspi_transfer_state

LPSPi transfer state, which is used for LPSPi transactional API state machine.

Values:

enumerator kLPSPi_Idle

Nothing in the transmitter/receiver.

enumerator kLPSPi_Busy

Transfer queue is not finished.

enumerator kLPSPi_Error

Transfer error.

typedef enum _lpspi_master_slave_mode lpspi_master_slave_mode_t

LPSPi master or slave mode configuration.

typedef enum _lpspi_which_pcs_config lpspi_which_pcs_t

LPSPi Peripheral Chip Select (PCS) configuration (which PCS to configure).

typedef enum _lpspi_pcs_polarity_config lpspi_pcs_polarity_config_t

LPSPi Peripheral Chip Select (PCS) Polarity configuration.

typedef enum *_lpspi_clock_polarity* lpspi_clock_polarity_t

LPSPI clock polarity configuration.

typedef enum *_lpspi_clock_phase* lpspi_clock_phase_t

LPSPI clock phase configuration.

typedef enum *_lpspi_shift_direction* lpspi_shift_direction_t

LPSPI data shifter direction options.

typedef enum *_lpspi_host_request_select* lpspi_host_request_select_t

LPSPI Host Request select configuration.

typedef enum *_lpspi_match_config* lpspi_match_config_t

LPSPI Match configuration options.

typedef enum *_lpspi_pin_config* lpspi_pin_config_t

LPSPI pin (SDO and SDI) configuration.

typedef enum *_lpspi_data_out_config* lpspi_data_out_config_t

LPSPI data output configuration.

typedef enum *_lpspi_transfer_width* lpspi_transfer_width_t

LPSPI transfer width configuration.

typedef enum *_lpspi_delay_type* lpspi_delay_type_t

LPSPI delay type selection.

typedef struct *_lpspi_master_config* lpspi_master_config_t

LPSPI master configuration structure.

typedef struct *_lpspi_slave_config* lpspi_slave_config_t

LPSPI slave configuration structure.

typedef struct *_lpspi_master_handle* lpspi_master_handle_t

Forward declaration of the *_lpspi_master_handle* typedefs.

typedef struct *_lpspi_slave_handle* lpspi_slave_handle_t

Forward declaration of the *_lpspi_slave_handle* typedefs.

typedef void (*lpspi_master_transfer_callback_t)(LP_SPI_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)(LP_SPI_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.

2.46 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

2.47 LPTMR: Low-Power Timer

`void LPTMR_Init(LPTMR_Type *base, const lptmr_config_t *config)`

Ungates the LPTMR clock and configures the peripheral for a basic operation.

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

Parameters

- `base` – LPTMR peripheral base address
- `config` – A pointer to the LPTMR configuration structure.

`void LPTMR_Deinit(LPTMR_Type *base)`

Gates the LPTMR clock.

Parameters

- `base` – LPTMR peripheral base address

`void LPTMR_GetDefaultConfig(lptmr_config_t *config)`

Fills in the LPTMR configuration structure with default settings.

The default values are as follows.

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

- `config` – A pointer to the LPTMR configuration structure.

`static inline void LPTMR_EnableInterrupts(LPTMR_Type *base, uint32_t mask)`

Enables the selected LPTMR interrupts.

Parameters

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

```
static inline void LPTMR_DisableInterrupts(LPTMR_Type *base, uint32_t mask)
```

Disables the selected LPTMR interrupts.

Parameters

- `base` – LPTMR peripheral base address
- `mask` – The interrupts to disable. This is a logical OR of members of the enumeration `lptmr_interrupt_enable_t`.

```
static inline uint32_t LPTMR_GetEnabledInterrupts(LPTMR_Type *base)
```

Gets the enabled LPTMR interrupts.

Parameters

- `base` – LPTMR peripheral base address

Returns

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

```
static inline uint32_t LPTMR_GetStatusFlags(LPTMR_Type *base)
```

Gets the LPTMR status flags.

Parameters

- `base` – LPTMR peripheral base address

Returns

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

```
static inline void LPTMR_ClearStatusFlags(LPTMR_Type *base, uint32_t mask)
```

Clears the LPTMR status flags.

Parameters

- `base` – LPTMR peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `lptmr_status_flags_t`.

```
static inline void LPTMR_SetTimerPeriod(LPTMR_Type *base, uint32_t ticks)
```

Sets the timer period in units of count.

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note:

- a. The TCF flag is set with the CNR equals the count provided here and then increments.
 - b. Call the utility macros provided in the `fsl_common.h` to convert to ticks.
-

Parameters

- `base` – LPTMR peripheral base address
- `ticks` – A timer period in units of ticks

```
static inline uint32_t LPTMR_GetCurrentTimerCount(LPTMR_Type *base)
```

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – LPTMR peripheral base address

Returns

The current counter value in ticks

static inline void LPTMR_StartTimer(LPTMR_Type *base)

Starts the timer.

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

- base – LPTMR peripheral base address

static inline void LPTMR_StopTimer(LPTMR_Type *base)

Stops the timer.

This function stops the timer and resets the timer's counter register.

Parameters

- base – LPTMR peripheral base address

FSL_LPTMR_DRIVER_VERSION

Driver Version

enum _lptmr_pin_select

LPTMR pin selection used in pulse counter mode.

Values:

enumerator kLPTMR_PinSelectInput_0

Pulse counter input 0 is selected

enumerator kLPTMR_PinSelectInput_1

Pulse counter input 1 is selected

enumerator kLPTMR_PinSelectInput_2

Pulse counter input 2 is selected

enumerator kLPTMR_PinSelectInput_3

Pulse counter input 3 is selected

enum _lptmr_pin_polarity

LPTMR pin polarity used in pulse counter mode.

Values:

enumerator kLPTMR_PinPolarityActiveHigh

Pulse Counter input source is active-high

enumerator kLPTMR_PinPolarityActiveLow

Pulse Counter input source is active-low

enum _lptmr_timer_mode

LPTMR timer mode selection.

Values:

enumerator kLPTMR_TimerModeTimeCounter

Time Counter mode

enumerator kLPTMR_TimerModePulseCounter

Pulse Counter mode

enum _lptmr_prescaler_glitch_value

LPTMR prescaler/glitch filter values.

Values:

enumerator kLPTMR_Prescale_Glitch_0

Prescaler divide 2, glitch filter does not support this setting

enumerator kLPTMR_Prescale_Glitch_1

Prescaler divide 4, glitch filter 2

enumerator kLPTMR_Prescale_Glitch_2

Prescaler divide 8, glitch filter 4

enumerator kLPTMR_Prescale_Glitch_3

Prescaler divide 16, glitch filter 8

enumerator kLPTMR_Prescale_Glitch_4

Prescaler divide 32, glitch filter 16

enumerator kLPTMR_Prescale_Glitch_5

Prescaler divide 64, glitch filter 32

enumerator kLPTMR_Prescale_Glitch_6

Prescaler divide 128, glitch filter 64

enumerator kLPTMR_Prescale_Glitch_7

Prescaler divide 256, glitch filter 128

enumerator kLPTMR_Prescale_Glitch_8

Prescaler divide 512, glitch filter 256

enumerator kLPTMR_Prescale_Glitch_9

Prescaler divide 1024, glitch filter 512

enumerator kLPTMR_Prescale_Glitch_10

Prescaler divide 2048 glitch filter 1024

enumerator kLPTMR_Prescale_Glitch_11

Prescaler divide 4096, glitch filter 2048

enumerator kLPTMR_Prescale_Glitch_12

Prescaler divide 8192, glitch filter 4096

enumerator kLPTMR_Prescale_Glitch_13

Prescaler divide 16384, glitch filter 8192

enumerator kLPTMR_Prescale_Glitch_14

Prescaler divide 32768, glitch filter 16384

enumerator kLPTMR_Prescale_Glitch_15

Prescaler divide 65536, glitch filter 32768

enum _lptmr_prescaler_clock_select

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

Values:

enum `_lptmr_interrupt_enable`

List of the LPTMR interrupts.

Values:

enumerator `kLPTMR_TimerInterruptEnable`

Timer interrupt enable

enum `_lptmr_status_flags`

List of the LPTMR status flags.

Values:

enumerator `kLPTMR_TimerCompareFlag`

Timer compare flag

typedef enum `_lptmr_pin_select` `lptmr_pin_select_t`

LPTMR pin selection used in pulse counter mode.

typedef enum `_lptmr_pin_polarity` `lptmr_pin_polarity_t`

LPTMR pin polarity used in pulse counter mode.

typedef enum `_lptmr_timer_mode` `lptmr_timer_mode_t`

LPTMR timer mode selection.

typedef enum `_lptmr_prescaler_glitch_value` `lptmr_prescaler_glitch_value_t`

LPTMR prescaler/glitch filter values.

typedef enum `_lptmr_prescaler_clock_select` `lptmr_prescaler_clock_select_t`

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

typedef enum `_lptmr_interrupt_enable` `lptmr_interrupt_enable_t`

List of the LPTMR interrupts.

typedef enum `_lptmr_status_flags` `lptmr_status_flags_t`

List of the LPTMR status flags.

typedef struct `_lptmr_config` `lptmr_config_t`

LPTMR config structure.

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

The configuration struct can be made constant so it resides in flash.

static inline void `LPTMR_EnableTimerDMA(LPTMR_Type *base, bool enable)`

Enable or disable timer DMA request.

Parameters

- `base` – base LPTMR peripheral base address
- `enable` – Switcher of timer DMA feature. “true” means to enable, “false” means to disable.

struct `_lptmr_config`

`#include <fsl_lptmr.h>` LPTMR config structure.

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

The configuration struct can be made constant so it resides in flash.

Public Members

lptmr_timer_mode_t timerMode

Time counter mode or pulse counter mode

lptmr_pin_select_t pinSelect

LPTMR pulse input pin select; used only in pulse counter mode

lptmr_pin_polarity_t pinPolarity

LPTMR pulse input pin polarity; used only in pulse counter mode

bool enableFreeRunning

True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set

bool bypassPrescaler

True: bypass prescaler; false: use clock from prescaler

lptmr_prescaler_clock_select_t prescalerClockSource

LPTMR clock source

lptmr_prescaler_glitch_value_t value

Prescaler or glitch filter value

2.48 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.49 LPUART Driver

static inline void LPUART_SoftwareReset(LPUART_Type *base)

Resets the LPUART using software.

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

- base – LPUART peripheral base address.

status_t LPUART_Init(LPUART_Type *base, const *lpuart_config_t* *config, uint32_t srcClock_Hz)

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefaultConfig() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
```

(continues on next page)

(continued from previous page)

```
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

```
void 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.

```
void LPUART_GetDefaultConfig(lpuart_config_t *config)
```

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

```
status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t
                             srcClock_Hz)
```

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus_Success – Set baudrate succeeded.


```
void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)
```

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, false to disable.

```
static inline void LPUART_SetMatchAddress(LPUART_Type *base, uint16_t address1, uint16_t  
                                         address2)
```

Set the LPUART address.

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – LPUART peripheral base address.
- address1 – LPUART slave address1.
- address2 – LPUART slave address2.

```
static inline void LPUART_EnableMatchAddress(LPUART_Type *base, bool match1, bool  
                                             match2)
```

Enable the LPUART match address feature.

Parameters

- base – LPUART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

```
static inline void LPUART_SetRxFifoWatermark(LPUART_Type *base, uint8_t water)
```

Sets the rx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Rx FIFO watermark.

```
static inline void LPUART_SetTxFifoWatermark(LPUART_Type *base, uint8_t water)
```

Sets the tx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Tx FIFO watermark.

static inline void LPUART_TransferEnable16Bit(*lpuart_handle_t* *handle, bool enable)

Sets the LPUART using 16bit transmit, only for 9bit or 10bit mode.

This function Enable 16bit Data transmit in lpuart_handle_t.

Parameters

- handle – LPUART handle pointer.
- enable – true to enable, false to disable.

uint32_t LPUART_GetStatusFlags(LPUART_Type *base)

Gets LPUART status flags.

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators `_lpuart_flags`. To check for a specific status, compare the return value with enumerators in the `_lpuart_flags`. For example, to check whether the TX is empty:

```
if (kLPUART_TxDataRegEmptyFlag & LPUART_GetStatusFlags(LPUART1))
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART status flags which are Ored by the enumerators in the `_lpuart_flags`.

status_t LPUART_ClearStatusFlags(LPUART_Type *base, uint32_t mask)

Clears status flags with a provided mask.

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only be cleared or set by hardware are: `kLPUART_TxDataRegEmptyFlag`, `kLPUART_TransmissionCompleteFlag`, `kLPUART_RxDataRegFullFlag`, `kLPUART_RxActiveFlag`, `kLPUART_NoiseErrorFlag`, `kLPUART_ParityErrorFlag`, `kLPUART_TxFifoEmptyFlag`, `kLPUART_RxFifoEmptyFlag`. Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- base – LPUART peripheral base address.
- mask – the status flags to be cleared. The user can use the enumerators in the `_lpuart_status_flag_t` to do the OR operation and get the mask.

Return values

- `kStatus_LPUART_FlagCannotClearManually` – The flag can't be cleared by this function but it is cleared automatically by hardware.
- `kStatus_Success` – Status in the mask are cleared.

Returns

0 succeed, others failed.

void LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the `_lpuart_interrupt_enable`. This 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 __unnamed113__

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 __unnamed115__

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 __unnamed117__

Public Members

uint8_t *volatile rxData
Address of remaining data to receive.
uint16_t *volatile rxData16
Address of remaining data to receive.
union __unnamed119__

Public Members

uint8_t *rxRingBuffer
Start address of the receiver ring buffer.
uint16_t *rxRingBuffer16
Start address of the receiver ring buffer.

2.50 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

2.51 MCM: Miscellaneous Control Module

FSL_MCM_DRIVER_VERSION

MCM driver version.

Enum _mcm_interrupt_flag. Interrupt status flag mask. .

Values:

enumerator kMCM_CacheWriteBuffer

Cache Write Buffer Error Enable.

enumerator kMCM_ParityError

Cache Parity Error Enable.

enumerator kMCM_FPUInvalidOperation

FPU Invalid Operation Interrupt Enable.

enumerator kMCM_FPUDivideByZero

FPU Divide-by-zero Interrupt Enable.

enumerator kMCM_FPUOverflow

FPU Overflow Interrupt Enable.

enumerator kMCM_FPUUnderflow

FPU Underflow Interrupt Enable.

enumerator kMCM_FPUInexact

FPU Inexact Interrupt Enable.

enumerator kMCM_FPUInputDenormalInterrupt

FPU Input Denormal Interrupt Enable.

typedef union *_mcm_buffer_fault_attribute* mcm_buffer_fault_attribute_t

The union of buffer fault attribute.

typedef union *_mcm_lmem_fault_attribute* mcm_lmem_fault_attribute_t

The union of LMEM fault attribute.

static inline void MCM_EnableCrossbarRoundRobin(MCM_Type *base, bool enable)

Enables/Disables crossbar round robin.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable crossbar round robin.
 - **true** Enable crossbar round robin.
 - **false** disable crossbar round robin.

static inline void MCM_EnableInterruptStatus(MCM_Type *base, uint32_t mask)

Enables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(_mcm_interrupt_flag).

static inline void MCM_DisableInterruptStatus(MCM_Type *base, uint32_t mask)

Disables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(`_mcm_interrupt_flag`).

static inline uint16_t MCM_GetInterruptStatus(MCM_Type *base)

Gets the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline void MCM_ClearCacheWriteBufferErrorStatus(MCM_Type *base)

Clears the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline uint32_t MCM_GetBufferFaultAddress(MCM_Type *base)

Gets buffer fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetBufferFaultAttribute(MCM_Type *base, *mcm_buffer_fault_attribute_t* *bufferfault)

Gets buffer fault attributes.

Parameters

- base – MCM peripheral base address.

static inline uint32_t MCM_GetBufferFaultData(MCM_Type *base)

Gets buffer fault data.

Parameters

- base – MCM peripheral base address.

static inline void MCM_LimitCodeCachePeripheralWriteBuffering(MCM_Type *base, bool enable)

Limit code cache peripheral write buffering.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable limit code cache peripheral write buffering.
 - **true** Enable limit code cache peripheral write buffering.
 - **false** disable limit code cache peripheral write buffering.

static inline void MCM_BypassFixedCodeCacheMap(MCM_Type *base, bool enable)

Bypass fixed code cache map.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable bypass fixed code cache map.
 - **true** Enable bypass fixed code cache map.
 - **false** disable bypass fixed code cache map.

static inline void MCM_EnableCodeBusCache(MCM_Type *base, bool enable)

Enables/Disables code bus cache.

Parameters

- base – MCM peripheral base address.
- enable – Used to disable/enable code bus cache.
 - **true** Enable code bus cache.
 - **false** disable code bus cache.

static inline void MCM_ForceCodeCacheToNoAllocation(MCM_Type *base, bool enable)

Force code cache to no allocation.

Parameters

- base – MCM peripheral base address.
- enable – Used to force code cache to allocation or no allocation.
 - **true** Force code cache to no allocation.
 - **false** Force code cache to allocation.

static inline void MCM_EnableCodeCacheWriteBuffer(MCM_Type *base, bool enable)

Enables/Disables code cache write buffer.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable code cache write buffer.
 - **true** Enable code cache write buffer.
 - **false** Disable code cache write buffer.

static inline void MCM_ClearCodeBusCache(MCM_Type *base)

Clear code bus cache.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnablePcParityFaultReport(MCM_Type *base, bool enable)

Enables/Disables PC Parity Fault Report.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity Fault Report.
 - **true** Enable PC Parity Fault Report.
 - **false** disable PC Parity Fault Report.

static inline void MCM_EnablePcParity(MCM_Type *base, bool enable)

Enables/Disables PC Parity.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity.
 - **true** Enable PC Parity.
 - **false** disable PC Parity.

```
static inline void MCM_LockConfigState(MCM_Type *base)
```

Lock the configuration state.

Parameters

- base – MCM peripheral base address.

```
static inline void MCM_EnableCacheParityReporting(MCM_Type *base, bool enable)
```

Enables/Disables cache parity reporting.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable cache parity reporting.
 - **true** Enable cache parity reporting.
 - **false** disable cache parity reporting.

```
static inline uint32_t MCM_GetLmemFaultAddress(MCM_Type *base)
```

Gets LMEM fault address.

Parameters

- base – MCM peripheral base address.

```
static inline void MCM_GetLmemFaultAttribute(MCM_Type *base, mcm_lmem_fault_attribute_t  
*lmemFault)
```

Get LMEM fault attributes.

Parameters

- base – MCM peripheral base address.

```
static inline uint64_t MCM_GetLmemFaultData(MCM_Type *base)
```

Gets LMEM fault data.

Parameters

- base – MCM peripheral base address.

MCM_LMFATR_TYPE_MASK

MCM_LMFATR_MODE_MASK

MCM_LMFATR_BUFF_MASK

MCM_LMFATR_CACH_MASK

MCM_ISCR_STAT_MASK

FSL_COMPONENT_ID

```
union _mcm_buffer_fault_attribute
```

#include <fsl_mcm.h> The union of buffer fault attribute.

Public Members

```
uint32_t attribute
```

Indicates the faulting attributes, when a properly-enabled cache write buffer error interrupt event is detected.

```
struct _mcm_buffer_fault_attribute, _mcm_buffer_fault_attrib attribute_memory
```

```
struct _mcm_buffer_fault_attrib
```

#include <fsl_mcm.h>

Public Members

uint32_t busErrorDataAccessType

Indicates the type of cache write buffer access.

uint32_t busErrorPrivilegeLevel

Indicates the privilege level of the cache write buffer access.

uint32_t busErrorSize

Indicates the size of the cache write buffer access.

uint32_t busErrorAccess

Indicates the type of system bus access.

uint32_t busErrorMasterID

Indicates the crossbar switch bus master number of the captured cache write buffer bus error.

uint32_t busErrorOverrun

Indicates if another cache write buffer bus error is detected.

union _mcm_lmem_fault_attribute

#include <fsl_mcm.h> The union of LMEM fault attribute.

Public Members

uint32_t attribute

Indicates the attributes of the LMEM fault detected.

struct _mcm_lmem_fault_attribute._mcm_lmem_fault_attribut attribute__memory

struct _mcm_lmem_fault_attribut

#include <fsl_mcm.h>

Public Members

uint32_t parityFaultProtectionSignal

Indicates the features of parity fault protection signal.

uint32_t parityFaultMasterSize

Indicates the parity fault master size.

uint32_t parityFaultWrite

Indicates the parity fault is caused by read or write.

uint32_t backdoorAccess

Indicates the LMEM access fault is initiated by core access or backdoor access.

uint32_t parityFaultSyndrome

Indicates the parity fault syndrome.

uint32_t overrun

Indicates the number of faultss.

2.52 MIPI DSI Driver

void DSI_Init(MIPI_DSI_HOST_Type *base, const *dsi_config_t* *config)

Initializes an MIPI DSI host with the user configuration.

This function initializes the MIPI DSI host with the configuration, it should be called first before other MIPI DSI driver functions.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to a user-defined configuration structure.

void DSI_Deinit(MIPI_DSI_HOST_Type *base)

Deinitializes an MIPI DSI host.

This function should be called after all bother MIPI DSI driver functions.

Parameters

- base – MIPI DSI host peripheral base address.

void DSI_GetDefaultConfig(*dsi_config_t* *config)

Get the default configuration to initialize the MIPI DSI host.

The default value is:

```
config->numLanes = 4;
config->enableNonContinuousHsClk = false;
config->enableTxUlps = false;
config->autoInsertEoTp = true;
config->numExtraEoTp = 0;
config->htxTo_ByteClk = 0;
config->lrxHostTo_ByteClk = 0;
config->btaTo_ByteClk = 0;
```

Parameters

- config – Pointer to a user-defined configuration structure.

void DSI_SetDpiConfig(MIPI_DSI_HOST_Type *base, const *dsi_dpi_config_t* *config, uint8_t numLanes, uint32_t dpiPixelClkFreq_Hz, uint32_t dsiHsBitClkFreq_Hz)

Configure the DPI interface core.

This function sets the DPI interface configuration, it should be used in video mode.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to the DPI interface configuration.
- numLanes – Lane number, should be same with the setting in *dsi_dpi_config_t*.
- dpiPixelClkFreq_Hz – The DPI pixel clock frequency in Hz.
- dsiHsBitClkFreq_Hz – The DSI high speed bit clock frequency in Hz. It is the same with DPHY PLL output.

uint32_t DSI_InitDphy(MIPI_DSI_HOST_Type *base, const *dsi_dphy_config_t* *config, uint32_t refClkFreq_Hz)

Initializes the D-PHY.

This function configures the D-PHY timing and setups the D-PHY PLL based on user configuration. The configuration structure could be got by the function *DSI_GetDphyDefaultConfig*.

For some platforms there is not dedicated D-PHY PLL, indicated by the macro *FSL_FEATURE_MIPI_DSI_NO_DPHY_PLL*. For these platforms, the *refClkFreq_Hz* is useless.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to the D-PHY configuration.
- refClkFreq_Hz – The REFCLK frequency in Hz.

Returns

The actual D-PHY PLL output frequency. If could not configure the PLL to the target frequency, the return value is 0.

```
void DSI_DeinitDphy(MIPI_DSI_HOST_Type *base)
```

Deinitializes the D-PHY.

Power down the D-PHY PLL and shut down D-PHY.

Parameters

- base – MIPI DSI host peripheral base address.

```
void DSI_GetDphyDefaultConfig(dsi_dphy_config_t *config, uint32_t txHsBitClk_Hz, uint32_t txEscClk_Hz)
```

Get the default D-PHY configuration.

Gets the default D-PHY configuration, the timing parameters are set according to D-PHY specification. User could use the configuration directly, or change some parameters according to the special device.

Parameters

- config – Pointer to the D-PHY configuration.
- txHsBitClk_Hz – High speed bit clock in Hz.
- txEscClk_Hz – Esc clock in Hz.

```
static inline void DSI_EnableInterrupts(MIPI_DSI_HOST_Type *base, uint32_t intGroup1, uint32_t intGroup2)
```

Enable the interrupts.

The interrupts to enable are passed in as OR'ed mask value of _dsi_interrupt.

Parameters

- base – MIPI DSI host peripheral base address.
- intGroup1 – Interrupts to enable in group 1.
- intGroup2 – Interrupts to enable in group 2.

```
static inline void DSI_DisableInterrupts(MIPI_DSI_HOST_Type *base, uint32_t intGroup1, uint32_t intGroup2)
```

Disable the interrupts.

The interrupts to disable are passed in as OR'ed mask value of _dsi_interrupt.

Parameters

- base – MIPI DSI host peripheral base address.
- intGroup1 – Interrupts to disable in group 1.
- intGroup2 – Interrupts to disable in group 2.

```
static inline void DSI_GetAndClearInterruptStatus(MIPI_DSI_HOST_Type *base, uint32_t *intGroup1, uint32_t *intGroup2)
```

Get and clear the interrupt status.

Parameters

- base – MIPI DSI host peripheral base address.
- intGroup1 – Group 1 interrupt status.
- intGroup2 – Group 2 interrupt status.

```
static inline void DSI_SetDbiPixelFifoSendLevel(MIPI_DSI_HOST_Type *base, uint16_t
                                              sendLevel)
```

Configure the DBI pixel FIFO send level.

This controls the level at which the DBI Host bridge begins sending pixels

Parameters

- base – MIPI DSI host peripheral base address.
- sendLevel – Send level value set to register.

```
static inline void DSI_SetDbiPixelPayloadSize(MIPI_DSI_HOST_Type *base, uint16_t payloadSize)
```

Configure the DBI pixel payload size.

Configures maximum number of pixels that should be sent as one DSI packet. Recommended to be evenly divisible by the line size (in pixels).

Parameters

- base – MIPI DSI host peripheral base address.
- payloadSize – Payload size value set to register.

```
void DSI_SetApbPacketControl(MIPI_DSI_HOST_Type *base, uint16_t wordCount, uint8_t
                             virtualChannel, dsi_tx_data_type_t dataType, uint8_t flags)
```

Configure the APB packet to send.

This function configures the next APB packet transfer. After configuration, the packet transfer could be started with function `DSI_SendApbPacket`. If the packet is long packet, Use `DSI_WriteApbTxPayload` to fill the payload before start transfer.

Parameters

- base – MIPI DSI host peripheral base address.
- wordCount – For long packet, this is the byte count of the payload. For short packet, this is $(data1 \ll 8) | data0$.
- virtualChannel – Virtual channel.
- dataType – The packet data type, (DI).
- flags – The transfer control flags, see `_dsi_transfer_flags`.

```
void DSI_WriteApbTxPayload(MIPI_DSI_HOST_Type *base, const uint8_t *payload, uint16_t
                           payloadSize)
```

Fill the long APB packet payload.

Write the long packet payload to TX FIFO.

Parameters

- base – MIPI DSI host peripheral base address.
- payload – Pointer to the payload.
- payloadSize – Payload size in byte.

```
void DSI_WriteApbTxPayloadExt(MIPI_DSI_HOST_Type *base, const uint8_t *payload, uint16_t
                              payloadSize, bool sendDcsCmd, uint8_t dcsCmd)
```

Extended function to fill the payload to TX FIFO.

Write the long packet payload to TX FIFO. This function could be used in two ways

- a. Include the DCS command in parameter `payload`. In this case, the DCS command is the first byte of `payload`. The parameter `sendDcsCmd` is set to `false`, the `dcsCmd` is not used. This function is the same as `DSI_WriteApbTxPayload` when used in this way.
- b. The DCS command is not in parameter `payload`, but specified by parameter `dcsCmd`. In this case, the parameter `sendDcsCmd` is set to `true`, the `dcsCmd` is the DCS command to send. The `payload` is sent after `dcsCmd`.

Parameters

- `base` – MIPI DSI host peripheral base address.
- `payload` – Pointer to the payload.
- `payloadSize` – Payload size in byte.
- `sendDcsCmd` – If set to `true`, the DCS command is specified by `dcsCmd`, otherwise the DCS command is included in the `payload`.
- `dcsCmd` – The DCS command to send, only used when `sendDcsCmd` is `true`.

```
void DSI_ReadApbRxPayload(MIPI_DSI_HOST_Type *base, uint8_t *payload, uint16_t
                          payloadSize)
```

Read the long APB packet payload.

Read the long packet payload from RX FIFO. This function reads directly but does not check the RX FIFO status. Upper layer should make sure there are available data.

Parameters

- `base` – MIPI DSI host peripheral base address.
- `payload` – Pointer to the payload.
- `payloadSize` – Payload size in byte.

```
static inline void DSI_SendApbPacket(MIPI_DSI_HOST_Type *base)
```

Trigger the controller to send out APB packet.

Send the packet set by `DSI_SetApbPacketControl`.

Parameters

- `base` – MIPI DSI host peripheral base address.

```
static inline uint32_t DSI_GetApbStatus(MIPI_DSI_HOST_Type *base)
```

Get the APB status.

The return value is OR'ed value of `_dsi_apb_status`.

Parameters

- `base` – MIPI DSI host peripheral base address.

Returns

The APB status.

```
static inline uint32_t DSI_GetRxErrorStatus(MIPI_DSI_HOST_Type *base)
```

Get the error status during data transfer.

The return value is OR'ed value of `_dsi_rx_error_status`.

Parameters

- `base` – MIPI DSI host peripheral base address.

Returns

The error status.

static inline uint8_t DSI_GetEccRxErrorPosition(uint32_t rxErrorStatus)

Get the one-bit RX ECC error position.

When one-bit ECC RX error detected using DSI_GetRxErrorStatus, this function could be used to get the error bit position.

```
uint8_t eccErrorPos;
uint32_t rxErrorStatus = DSI_GetRxErrorStatus(MIPI_DSI);
if (kDSI_RxErrorEccOneBit & rxErrorStatus)
{
    eccErrorPos = DSI_GetEccRxErrorPosition(rxErrorStatus);
}
```

Parameters

- rxErrorStatus – The error status returned by DSI_GetRxErrorStatus.

Returns

The 1-bit ECC error position.

static inline uint32_t DSI_GetAndClearHostStatus(MIPI_DSI_HOST_Type *base)

Get and clear the DSI host status.

The host status are returned as mask value of _dsi_host_status.

Parameters

- base – MIPI DSI host peripheral base address.

Returns

The DSI host status.

static inline uint32_t DSI_GetRxPacketHeader(MIPI_DSI_HOST_Type *base)

Get the RX packet header.

Parameters

- base – MIPI DSI host peripheral base address.

Returns

The RX packet header.

static inline *dsi_rx_data_type_t* DSI_GetRxPacketType(uint32_t rxPktHeader)

Extract the RX packet type from the packet header.

Extract the RX packet type from the packet header get by DSI_GetRxPacketHeader.

Parameters

- rxPktHeader – The RX packet header get by DSI_GetRxPacketHeader.

Returns

The RX packet type.

static inline uint16_t DSI_GetRxPacketWordCount(uint32_t rxPktHeader)

Extract the RX packet word count from the packet header.

Extract the RX packet word count from the packet header get by DSI_GetRxPacketHeader.

Parameters

- rxPktHeader – The RX packet header get by DSI_GetRxPacketHeader.

Returns

For long packet, return the payload word count (byte). For short packet, return the (data0 « 8) | data1.

static inline uint8_t DSI_GetRxPacketVirtualChannel(uint32_t rxPktHeader)

Extract the RX packet virtual channel from the packet header.

Extract the RX packet virtual channel from the packet header get by DSI_GetRxPacketHeader.

Parameters

- rxPktHeader – The RX packet header get by DSI_GetRxPacketHeader.

Returns

The virtual channel.

status_t DSI_TransferBlocking(MIPI_DSI_HOST_Type *base, dsi_transfer_t *xfer)

APB data transfer using blocking method.

Perform APB data transfer using blocking method. This function waits until all data send or received, or timeout happens.

When using this API to read data, the actually read data count could be got from xfer->rxDataSize.

Parameters

- base – MIPI DSI host peripheral base address.
- xfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Data transfer finished with no error.
- kStatus_Timeout – Transfer failed because of timeout.
- kStatus_DSI_RxDataError – RX data error, user could use DSI_GetRxErrorStatus to check the error details.
- kStatus_DSI_ErrorReportReceived – Error Report packet received, user could use DSI_GetAndClearHostStatus to check the error report status.
- kStatus_DSI_NotSupported – Transfer format not supported.
- kStatus_DSI_Fail – Transfer failed for other reasons.

status_t DSI_TransferCreateHandle(MIPI_DSI_HOST_Type *base, dsi_handle_t *handle, dsi_callback_t callback, void *userData)

Create the MIPI DSI handle.

This function initializes the MIPI DSI handle which can be used for other transactional APIs.

Parameters

- base – MIPI DSI host peripheral base address.
- handle – Handle pointer.
- callback – Callback function.
- userData – User data.

status_t DSI_TransferNonBlocking(MIPI_DSI_HOST_Type *base, dsi_handle_t *handle, dsi_transfer_t *xfer)

APB data transfer using interrupt method.

Perform APB data transfer using interrupt method, when transfer finished, upper layer could be informed through callback function.

When using this API to read data, the actually read data count could be got from handle->xfer->rxDataSize after read finished.

Parameters

- base – MIPI DSI host peripheral base address.
- handle – pointer to dsi_handle_t structure which stores the transfer state.
- xfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Data transfer started successfully.
- kStatus_DSI_Busy – Failed to start transfer because DSI is busy with previous transfer.
- kStatus_DSI_NotSupported – Transfer format not supported.

void DSI_TransferAbort(MIPI_DSI_HOST_Type *base, dsi_handle_t *handle)

Abort current APB data transfer.

Parameters

- base – MIPI DSI host peripheral base address.
- handle – pointer to dsi_handle_t structure which stores the transfer state.

void DSI_TransferHandleIRQ(MIPI_DSI_HOST_Type *base, dsi_handle_t *handle)

Interrupt handler for the DSI.

Parameters

- base – MIPI DSI host peripheral base address.
- handle – pointer to dsi_handle_t structure which stores the transfer state.

FSL_MIPI_DSI_DRIVER_VERSION

Error codes for the MIPI DSI driver.

Values:

enumerator kStatus_DSI_Busy

DSI is busy.

enumerator kStatus_DSI_RxDataError

Read data error.

enumerator kStatus_DSI_ErrorReportReceived

Error report package received.

enumerator kStatus_DSI_NotSupported

The transfer type not supported.

enum _dsi_dpi_color_coding

MIPI DPI interface color coding.

Values:

enumerator kDSI_Dpi16BitConfig1

16-bit configuration 1. RGB565: XXXXXXXX_RRRRRGGG_GGGBBBBB.

enumerator kDSI_Dpi16BitConfig2

16-bit configuration 2. RGB565: XXXRRRRR_XXGGGGGG_XXXBBBBB.

enumerator kDSI_Dpi16BitConfig3

16-bit configuration 3. RGB565: XXRRRRRX_XXGGGGGG_XXBBBBBX.

enumerator kDSI_Dpi18BitConfig1

18-bit configuration 1. RGB666: XXXXXXRR_RRRRGGGG_GGBBBBBB.

enumerator kDSI_Dpi18BitConfig2

18-bit configuration 2. RGB666: XXRRRRRRR_XXGGGGGGG_XXBBBBBBB.

enumerator kDSI_Dpi24Bit

24-bit.

enum _dsi_dpi_pixel_packet

MIPI DSI pixel packet type send through DPI interface.

Values:

enumerator kDSI_PixelPacket16Bit

16 bit RGB565.

enumerator kDSI_PixelPacket18Bit

18 bit RGB666 packed.

enumerator kDSI_PixelPacket18BitLoosely

18 bit RGB666 loosely packed into three bytes.

enumerator kDSI_PixelPacket24Bit

24 bit RGB888, each pixel uses three bytes.

_dsi_dpi_polarity_flag DPI signal polarity.

Values:

enumerator kDSI_DpiVsyncActiveLow

VSYNC active low.

enumerator kDSI_DpiHsyncActiveLow

HSYNC active low.

enumerator kDSI_DpiVsyncActiveHigh

VSYNC active high.

enumerator kDSI_DpiHsyncActiveHigh

HSYNC active high.

enum _dsi_dpi_video_mode

DPI video mode.

Values:

enumerator kDSI_DpiNonBurstWithSyncPulse

Non-Burst mode with Sync Pulses.

enumerator kDSI_DpiNonBurstWithSyncEvent

Non-Burst mode with Sync Events.

enumerator kDSI_DpiBurst

Burst mode.

enum _dsi_dpi_bllp_mode

Behavior in BLLP (Blanking or Low-Power Interval).

Values:

enumerator kDSI_DpiBllpLowPower

LP mode used in BLLP periods.

enumerator kDSI_DpiBllpBlanking

Blanking packets used in BLLP periods.

enumerator kDSI_DpiBllpNull
Null packets used in BLLP periods.

_dsi_apb_status Status of APB to packet interface.

Values:

enumerator kDSI_ApbNotIdle
State machine not idle

enumerator kDSI_ApbTxDone
Tx packet done

enumerator kDSI_ApbRxControl
DPHY direction 0 - tx had control, 1 - rx has control

enumerator kDSI_ApbTxOverflow
TX fifo overflow

enumerator kDSI_ApbTxUnderflow
TX fifo underflow

enumerator kDSI_ApbRxOverflow
RX fifo overflow

enumerator kDSI_ApbRxUnderflow
RX fifo underflow

enumerator kDSI_ApbRxHeaderReceived
RX packet header has been received

enumerator kDSI_ApbRxPacketReceived
All RX packet payload data has been received

_dsi_rx_error_status Host receive error status.

Values:

enumerator kDSI_RxErrorEccOneBit
ECC single bit error detected.

enumerator kDSI_RxErrorEccMultiBit
ECC multi bit error detected.

enumerator kDSI_RxErrorCrc
CRC error detected.

enumerator kDSI_RxErrorHtxTo
High Speed forward TX timeout detected.

enumerator kDSI_RxErrorLrxTo
Reverse Low power data receive timeout detected.

enumerator kDSI_RxErrorBtaTo
BTA timeout detected.

enum _dsi_host_status
DSI host controller status (status_out)

Values:

enumerator kDSI_HostSoTError

SoT error from peripheral error report.

enumerator kDSI_HostSoTSyncError

SoT Sync error from peripheral error report.

enumerator kDSI_HostEoTSyncError

EoT Sync error from peripheral error report.

enumerator kDSI_HostEscEntryCmdError

Escape Mode Entry Command Error from peripheral error report.

enumerator kDSI_HostLpTxSyncError

Low-power transmit Sync Error from peripheral error report.

enumerator kDSI_HostPeriphToError

Peripheral timeout error from peripheral error report.

enumerator kDSI_HostFalseControlError

False control error from peripheral error report.

enumerator kDSI_HostContentionDetected

Contention detected from peripheral error report.

enumerator kDSI_HostEccErrorOneBit

Single bit ECC error (corrected) from peripheral error report.

enumerator kDSI_HostEccErrorMultiBit

Multi bit ECC error (not corrected) from peripheral error report.

enumerator kDSI_HostChecksumError

Checksum error from peripheral error report.

enumerator kDSI_HostInvalidDataType

DSI data type not recognized.

enumerator kDSI_HostInvalidVcId

DSI VC ID invalid.

enumerator kDSI_HostInvalidTxLength

Invalid transmission length.

enumerator kDSI_HostProtocalViolation

DSI protocol violation.

enumerator kDSI_HostResetTriggerReceived

Reset trigger received.

enumerator kDSI_HostTearTriggerReceived

Tear effect trigger receive.

enumerator kDSI_HostAckTriggerReceived

Acknowledge trigger message received.

_dsi_interrupt DSI interrupt.

Values:

enumerator kDSI_InterruptGroup1ApbNotIdle

State machine not idle

enumerator kDSI_InterruptGroup1ApbTxDone
Tx packet done

enumerator kDSI_InterruptGroup1ApbRxControl
DPHY direction 0 - tx control, 1 - rx control

enumerator kDSI_InterruptGroup1ApbTxOverflow
TX fifo overflow

enumerator kDSI_InterruptGroup1ApbTxUnderflow
TX fifo underflow

enumerator kDSI_InterruptGroup1ApbRxOverflow
RX fifo overflow

enumerator kDSI_InterruptGroup1ApbRxUnderflow
RX fifo underflow

enumerator kDSI_InterruptGroup1ApbRxHeaderReceived
RX packet header has been received

enumerator kDSI_InterruptGroup1ApbRxPacketReceived
All RX packet payload data has been received

enumerator kDSI_InterruptGroup1SoTError
SoT error from peripheral error report.

enumerator kDSI_InterruptGroup1SoTSyncError
SoT Sync error from peripheral error report.

enumerator kDSI_InterruptGroup1EoTSyncError
EoT Sync error from peripheral error report.

enumerator kDSI_InterruptGroup1EscEntryCmdError
Escape Mode Entry Command Error from peripheral error report.

enumerator kDSI_InterruptGroup1LpTxSyncError
Low-power transmit Sync Error from peripheral error report.

enumerator kDSI_InterruptGroup1PeriphToError
Peripheral timeout error from peripheral error report.

enumerator kDSI_InterruptGroup1FalseControlError
False control error from peripheral error report.

enumerator kDSI_InterruptGroup1ContentionDetected
Contention detected from peripheral error report.

enumerator kDSI_InterruptGroup1EccErrorOneBit
Single bit ECC error (corrected) from peripheral error report.

enumerator kDSI_InterruptGroup1EccErrorMultiBit
Multi bit ECC error (not corrected) from peripheral error report.

enumerator kDSI_InterruptGroup1ChecksumError
Checksum error from peripheral error report.

enumerator kDSI_InterruptGroup1InvalidDataType
DSI data type not recognized.

enumerator kDSI_InterruptGroup1InvalidVcId
DSI VC ID invalid.

enumerator kDSI_InterruptGroup1InvalidTxLength

Invalid transmission length.

enumerator kDSI_InterruptGroup1ProtocalViolation

DSI protocol violation.

enumerator kDSI_InterruptGroup1ResetTriggerReceived

Reset trigger received.

enumerator kDSI_InterruptGroup1TearTriggerReceived

Tear effect trigger receive.

enumerator kDSI_InterruptGroup1AckTriggerReceived

Acknowledge trigger message received.

enumerator kDSI_InterruptGroup1HtxTo

High speed TX timeout.

enumerator kDSI_InterruptGroup1LrxTo

Low power RX timeout.

enumerator kDSI_InterruptGroup1BtaTo

Host BTA timeout.

enumerator kDSI_InterruptGroup2EccOneBit

Sinle bit ECC error.

enumerator kDSI_InterruptGroup2EccMultiBit

Multi bit ECC error.

enumerator kDSI_InterruptGroup2CrcError

CRC error.

enum _dsi_tx_data_type

DSI TX data type.

Values:

enumerator kDSI_TxDataVsyncStart

V Sync start.

enumerator kDSI_TxDataVsyncEnd

V Sync end.

enumerator kDSI_TxDataHsyncStart

H Sync start.

enumerator kDSI_TxDataHsyncEnd

H Sync end.

enumerator kDSI_TxDataEoTp

End of transmission packet.

enumerator kDSI_TxDataCmOff

Color mode off.

enumerator kDSI_TxDataCmOn

Color mode on.

enumerator kDSI_TxDataShutDownPeriph

Shut down peripheral.

enumerator kDSI_TxDataTurnOnPeriph
Turn on peripheral.

enumerator kDSI_TxDataGenShortWrNoParam
Generic Short WRITE, no parameters.

enumerator kDSI_TxDataGenShortWrOneParam
Generic Short WRITE, one parameter.

enumerator kDSI_TxDataGenShortWrTwoParam
Generic Short WRITE, two parameter.

enumerator kDSI_TxDataGenShortRdNoParam
Generic Short READ, no parameters.

enumerator kDSI_TxDataGenShortRdOneParam
Generic Short READ, one parameter.

enumerator kDSI_TxDataGenShortRdTwoParam
Generic Short READ, two parameter.

enumerator kDSI_TxDataDcsShortWrNoParam
DCS Short WRITE, no parameters.

enumerator kDSI_TxDataDcsShortWrOneParam
DCS Short WRITE, one parameter.

enumerator kDSI_TxDataDcsShortRdNoParam
DCS Short READ, no parameters.

enumerator kDSI_TxDataSetMaxReturnPktSize
Set the Maximum Return Packet Size.

enumerator kDSI_TxDataNull
Null Packet, no data.

enumerator kDSI_TxDataBlanking
Blanking Packet, no data.

enumerator kDSI_TxDataGenLongWr
Generic long write.

enumerator kDSI_TxDataDcsLongWr
DCS Long Write/write_LUT Command Packet.

enumerator kDSI_TxDataLooselyPackedPixel20BitYCbCr
Loosely Packed Pixel Stream, 20-bit YCbCr, 4:2:2 Format.

enumerator kDSI_TxDataPackedPixel24BitYCbCr
Packed Pixel Stream, 24-bit YCbCr, 4:2:2 Format.

enumerator kDSI_TxDataPackedPixel16BitYCbCr
Packed Pixel Stream, 16-bit YCbCr, 4:2:2 Format.

enumerator kDSI_TxDataPackedPixel30BitRGB
Packed Pixel Stream, 30-bit RGB, 10-10-10 Format.

enumerator kDSI_TxDataPackedPixel36BitRGB
Packed Pixel Stream, 36-bit RGB, 12-12-12 Format.

enumerator kDSI_TxDataPackedPixel12BitYCrCb
Packed Pixel Stream, 12-bit YCbCr, 4:2:0 Format.

enumerator kDSI_TxDataPackedPixel16BitRGB

Packed Pixel Stream, 16-bit RGB, 5-6-5 Format.

enumerator kDSI_TxDataPackedPixel18BitRGB

Packed Pixel Stream, 18-bit RGB, 6-6-6 Format.

enumerator kDSI_TxDataLooselyPackedPixel18BitRGB

Loosely Packed Pixel Stream, 18-bit RGB, 6-6-6 Format.

enumerator kDSI_TxDataPackedPixel24BitRGB

Packed Pixel Stream, 24-bit RGB, 8-8-8 Format.

enum _dsi_rx_data_type

DSI RX data type.

Values:

enumerator kDSI_RxDataAckAndErrorReport

Acknowledge and Error Report

enumerator kDSI_RxDataEoTp

End of Transmission packet.

enumerator kDSI_RxDataGenShortRdResponseOneByte

Generic Short READ Response, 1 byte returned.

enumerator kDSI_RxDataGenShortRdResponseTwoByte

Generic Short READ Response, 2 byte returned.

enumerator kDSI_RxDataGenLongRdResponse

Generic Long READ Response.

enumerator kDSI_RxDataDcsLongRdResponse

DCS Long READ Response.

enumerator kDSI_RxDataDcsShortRdResponseOneByte

DCS Short READ Response, 1 byte returned.

enumerator kDSI_RxDataDcsShortRdResponseTwoByte

DCS Short READ Response, 2 byte returned.

_dsi_transfer_flags DSI transfer control flags.

Values:

enumerator kDSI_TransferUseHighSpeed

Use high speed mode or not.

enumerator kDSI_TransferPerformBTA

Perform BTA or not.

typedef struct _dsi_config dsi_config_t

MIPI DSI controller configuration.

typedef enum _dsi_dpi_color_coding dsi_dpi_color_coding_t

MIPI DPI interface color coding.

typedef enum _dsi_dpi_pixel_packet dsi_dpi_pixel_packet_t

MIPI DSI pixel packet type send through DPI interface.

typedef enum _dsi_dpi_video_mode dsi_dpi_video_mode_t

DPI video mode.

typedef enum *_dsi_dpi_bllp_mode* dsi_dpi_bllp_mode_t

Behavior in BLLP (Blanking or Low-Power Interval).

typedef struct *_dsi_dpi_config* dsi_dpi_config_t

MIPI DSI controller DPI interface configuration.

typedef struct *_dsi_dphy_config* dsi_dphy_config_t

MIPI DSI D-PHY configuration.

typedef enum *_dsi_tx_data_type* dsi_tx_data_type_t

DSI TX data type.

typedef enum *_dsi_rx_data_type* dsi_rx_data_type_t

DSI RX data type.

typedef struct *_dsi_transfer* dsi_transfer_t

Structure for the data transfer.

typedef struct *_dsi_handle* dsi_handle_t

MIPI DSI transfer handle.

typedef void (*dsi_callback_t)(MIPI_DSI_HOST_Type *base, *dsi_handle_t* *handle, *status_t* status, void *userData)

MIPI DSI callback for finished transfer.

When transfer finished, one of these status values will be passed to the user:

- kStatus_Success Data transfer finished with no error.
- kStatus_Timeout Transfer failed because of timeout.
- kStatus_DSI_RxDataError RX data error, user could use DSI_GetRxErrorStatus to check the error details.
- kStatus_DSI_ErrorReportReceived Error Report packet received, user could use DSI_GetAndClearHostStatus to check the error report status.
- kStatus_Fail Transfer failed for other reasons.

FSL_DSI_TX_MAX_PAYLOAD_BYTE

FSL_DSI_RX_MAX_PAYLOAD_BYTE

struct *_dsi_config*

#include <fsl_mipi_dsi.h> MIPI DSI controller configuration.

Public Members

uint8_t numLanes

Number of lanes.

bool enableNonContinuousHsClk

In enabled, the high speed clock will enter low power mode between transmissions.

bool enableTxUlps

Enable the TX ULPS.

bool autoInsertEoTp

Insert an EoTp short package when switching from HS to LP.

uint8_t numExtraEoTp

How many extra EoTp to send after the end of a packet.

uint32_t htxTo_ByteClk

HS TX timeout count (HTX_TO) in byte clock.

uint32_t lrxHostTo_ByteClk

LP RX host timeout count (LRX-H_TO) in byte clock.

uint32_t btaTo_ByteClk

Bus turn around timeout count (TA_TO) in byte clock.

struct _dsi_dpi_config

#include <fsl_mipi_dsi.h> MIPI DSI controller DPI interface configuration.

Public Members

uint16_t pixelPayloadSize

Maximum number of pixels that should be sent as one DSI packet. Recommended that the line size (in pixels) is evenly divisible by this parameter.

dsi_dpi_color_coding_t dpiColorCoding

DPI color coding.

dsi_dpi_pixel_packet_t pixelPacket

Pixel packet format.

dsi_dpi_video_mode_t videoMode

Video mode.

dsi_dpi_bllp_mode_t bllpMode

Behavior in BLLP.

uint8_t polarityFlags

OR'ed value of _dsi_dpi_polarity_flag controls signal polarity.

uint16_t hfp

Horizontal front porch, in dpi pixel clock.

uint16_t hbp

Horizontal back porch, in dpi pixel clock.

uint16_t hsw

Horizontal sync width, in dpi pixel clock.

uint8_t vfp

Number of lines in vertical front porch.

uint8_t vbp

Number of lines in vertical back porch.

uint16_t panelHeight

Line number in vertical active area.

uint8_t virtualChannel

Virtual channel.

struct _dsi_dphy_config

#include <fsl_mipi_dsi.h> MIPI DSI D-PHY configuration.

Public Members

uint32_t txHsBitClk_Hz

The generated HS TX bit clock in Hz.

uint8_t tClkPre_ByteClk

TLPX + TCLK-PREPARE + TCLK-ZERO + TCLK-PRE in byte clock. Set how long the controller will wait after enabling clock lane for HS before enabling data lanes for HS.

uint8_t tClkPost_ByteClk

TCLK-POST + T_CLK-TRAIL in byte clock. Set how long the controller will wait before putting clock lane into LP mode after data lanes detected in stop state.

uint8_t tHsExit_ByteClk

THS-EXIT in byte clock. Set how long the controller will wait after the clock lane has been put into LP mode before enabling clock lane for HS again.

uint32_t tWakeup_EscClk

Number of clk_esc clock periods to keep a clock or data lane in Mark-1 state after exiting ULPS.

uint8_t tHsPrepare_HalfEscClk

THS-PREPARE in clk_esc/2. Set how long to drive the LP-00 state before HS transmissions, available values are 2, 3, 4, 5.

uint8_t tClkPrepare_HalfEscClk

TCLK-PREPARE in clk_esc/2. Set how long to drive the LP-00 state before HS transmissions, available values are 2, 3.

uint8_t tHsZero_ByteClk

THS-ZERO in clk_byte. Set how long that controller drives data lane HS-0 state before transmit the Sync sequence. Available values are 6, 7, ..., 37.

uint8_t tClkZero_ByteClk

TCLK-ZERO in clk_byte. Set how long that controller drives clock lane HS-0 state before transmit the Sync sequence. Available values are 3, 4, ..., 66.

uint8_t tHsTrail_ByteClk

THS-TRAIL + 4*UI in clk_byte. Set the time of the flipped differential state after last payload data bit of HS transmission burst. Available values are 0, 1, ..., 15.

uint8_t tClkTrail_ByteClk

TCLK-TRAIL + 4*UI in clk_byte. Set the time of the flipped differential state after last payload data bit of HS transmission burst. Available values are 0, 1, ..., 15.

struct _dsi_transfer

#include <fsl_mipi_dsi.h> Structure for the data transfer.

Public Members

uint8_t virtualChannel

Virtual channel.

dsi_tx_data_type_t txDataType

TX data type.

uint8_t flags

Flags to control the transfer, see _dsi_transfer_flags.

const uint8_t *txData

The TX data buffer.

uint8_t *rxData

The RX data buffer.

uint16_t txDataSize

Size of the TX data.

uint16_t rxDataSize

Size of the RX data.

bool sendDscCmd

If set to true, the DCS command is specified by dscCmd, otherwise the DCS command is included in the txData.

uint8_t dscCmd

The DCS command to send, only valid when sendDscCmd is true.

struct _dsi_handle

#include <fsl_mipi_dsi.h> MIPI DSI transfer handle structure.

Public Members

volatile bool isBusy

MIPI DSI is busy with APB data transfer.

dsi_transfer_t xfer

Transfer information.

dsi_callback_t callback

DSI callback

void *userData

Callback parameter

2.53 MIPI_DSI: MIPI DSI Host Controller

2.54 MRT: Multi-Rate Timer

void MRT_Init(MRT_Type *base, const *mrt_config_t* *config)

Ungates the MRT clock and configures the peripheral for basic operation.

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

Parameters

- base – Multi-Rate timer peripheral base address
- config – Pointer to user's MRT config structure. If MRT has MULTITASK bit field in MODCFG register, param config is useless.

void MRT_Deinit(MRT_Type *base)

Gate the MRT clock.

Parameters

- base – Multi-Rate timer peripheral base address

static inline void MRT_GetDefaultConfig(*mrt_config_t* *config)

Fill in the MRT config struct with the default settings.

The default values are:

```
config->enableMultiTask = false;
```

Parameters

- config – Pointer to user's MRT config structure.

static inline void MRT_SetupChannelMode(MRT_Type *base, *mrt_chnl_t* channel, const *mrt_timer_mode_t* mode)

Sets up an MRT channel mode.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Channel that is being configured.
- mode – Timer mode to use for the channel.

static inline void MRT_EnableInterrupts(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Enables the MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline void MRT_DisableInterrupts(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Disables the selected MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline uint32_t MRT_GetEnabledInterrupts(MRT_Type *base, *mrt_chnl_t* channel)

Gets the enabled MRT interrupts.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration *mrt_interrupt_enable_t*

static inline uint32_t MRT_GetStatusFlags(MRT_Type *base, *mrt_chnl_t* channel)

Gets the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

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

```
static inline void MRT_ClearStatusFlags(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Clears the MRT status flags.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `mrt_status_flags_t`

```
void MRT_UpdateTimerPeriod(MRT_Type *base, mrt_chnl_t channel, uint32_t count, bool  
                           immediateLoad)
```

Used to update the timer period in units of count.

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number
- `count` – Timer period in units of ticks
- `immediateLoad` – `true`: Load the new value immediately into the TIMER register; `false`: Load the new value at the end of current timer interval

```
static inline uint32_t MRT_GetCurrentTimerCount(MRT_Type *base, mrt_chnl_t channel)
```

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number

Returns

Current timer counting value in ticks

```
static inline void MRT_StartTimer(MRT_Type *base, mrt_chnl_t channel, uint32_t count)
```

Starts the timer counting.

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.
- `count` – Timer period in units of ticks. Count can contain the LOAD bit, which control the force load feature.

static inline void MRT_StopTimer(MRT_Type *base, *mrt_chnl_t* channel)

Stops the timer counting.

This function stops the timer from counting.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

static inline uint32_t MRT_GetIdleChannel(MRT_Type *base)

Find the available channel.

This function returns the lowest available channel number.

Parameters

- `base` – Multi-Rate timer peripheral base address

static inline void MRT_ReleaseChannel(MRT_Type *base, *mrt_chnl_t* channel)

Release the channel when the timer is using the multi-task mode.

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling `MRT_GetIdleChannel()` for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

FSL_MRT_DRIVER_VERSION

enum *_mrt_chnl*

List of MRT channels.

Values:

enumerator `kMRT_Channel_0`

MRT channel number 0

enumerator `kMRT_Channel_1`

MRT channel number 1

enumerator `kMRT_Channel_2`

MRT channel number 2

enumerator `kMRT_Channel_3`

MRT channel number 3

enum *_mrt_timer_mode*

List of MRT timer modes.

Values:

enumerator `kMRT_RepeatMode`

Repeat Interrupt mode

enumerator kMRT_OneShotMode

One-shot Interrupt mode

enumerator kMRT_OneShotStallMode

One-shot stall mode

enum `_mrt_interrupt_enable`

List of MRT interrupts.

Values:

enumerator kMRT_TimerInterruptEnable

Timer interrupt enable

enum `_mrt_status_flags`

List of MRT status flags.

Values:

enumerator kMRT_TimerInterruptFlag

Timer interrupt flag

enumerator kMRT_TimerRunFlag

Indicates state of the timer

typedef enum `_mrt_chnl` `mrt_chnl_t`

List of MRT channels.

typedef enum `_mrt_timer_mode` `mrt_timer_mode_t`

List of MRT timer modes.

typedef enum `_mrt_interrupt_enable` `mrt_interrupt_enable_t`

List of MRT interrupts.

typedef enum `_mrt_status_flags` `mrt_status_flags_t`

List of MRT status flags.

typedef struct `_mrt_config` `mrt_config_t`

MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the `MRT_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

struct `_mrt_config`

`#include <fsl_mrt.h>` MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the `MRT_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

bool `enableMultiTask`

true: Timers run in multi-task mode; false: Timers run in hardware status mode

2.55 MU: Messaging Unit

void MU_Init(MU_Type *base)

Initializes the MU module.

This function enables the MU clock only.

Parameters

- base – MU peripheral base address.

void MU_Deinit(MU_Type *base)

De-initializes the MU module.

This function disables the MU clock only.

Parameters

- base – MU peripheral base address.

static inline void MU_SendMsgNonBlocking(MU_Type *base, uint32_t regIndex, uint32_t msg)

Writes a message to the TX register.

This function writes a message to the specific TX register. It does not check whether the TX register is empty or not. The upper layer should make sure the TX register is empty before calling this function. This function can be used in ISR for better performance.

```
while (!(kMU_Tx0EmptyFlag & MU_GetStatusFlags(base))) { } Wait for TX0 register empty.  
MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG_VAL); Write message to the TX0 register.
```

Parameters

- base – MU peripheral base address.
- regIndex – TX register index, see `mu_msg_reg_index_t`.
- msg – Message to send.

status_t MU_SendMsg(MU_Type *base, uint32_t regIndex, uint32_t msg)

Blocks to send a message.

This function waits until the TX register is empty and sends the message. If `MU_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and returns `kStatus_Timeout`.

This function waits until the TX register is empty and sends the message. If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and returns `kStatus_Timeout`.

Parameters

- base – MU peripheral base address.
- regIndex – MU message register, see `mu_msg_reg_index_t`.
- msg – Message to send.
- base – MU peripheral base address.
- regIndex – MU message register, see `mu_msg_reg_index_t`.
- msg – Message to send.

Return values

- `kStatus_Success` – Message sent successfully.

- `kStatus_Timeout` – Timeout occurred while waiting for TX register to be empty.
- `kStatus_Success` – Message sent successfully.
- `kStatus_Timeout` – Timeout occurred while waiting for TX register to be empty.

Returns

`status_t`

Returns

`status_t`

`static inline uint32_t MU_ReceiveMsgNonBlocking(MU_Type *base, uint32_t regIndex)`

Reads a message from the RX register.

This function reads a message from the specific RX register. It does not check whether the RX register is full or not. The upper layer should make sure the RX register is full before calling this function. This function can be used in ISR for better performance.

```
uint32_t msg;
while (!(kMU_Rx0FullFlag & MU_GetStatusFlags(base)))
{
    } Wait for the RX0 register full.

msg = MU_ReceiveMsgNonBlocking(base, kMU_MsgReg0); Read message from RX0 register.
```

Parameters

- `base` – MU peripheral base address.
- `regIndex` – RX register index, see `mu_msg_reg_index_t`.

Returns

The received message.

`status_t MU_ReceiveMsgTimeout(MU_Type *base, uint32_t regIndex, uint32_t *readValue)`

Blocks to receive a message with timeout protection.

This function waits until the RX register is full and receives the message. If `MU_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout`.

This function provides the same blocking behavior as `MU_ReceiveMsg()` but with additional timeout protection to prevent system hangs if the other core becomes unresponsive or if hardware issues occur.

Note: Both `MU_ReceiveMsg()` and `MU_ReceiveMsgTimeout()` are blocking functions. The difference is that this function includes timeout protection while `MU_ReceiveMsg()` waits indefinitely.

Parameters

- `base` – MU peripheral base address.
- `regIndex` – RX register index, see `mu_msg_reg_index_t`.
- `readValue` – Pointer to store the received message.

Return values

- `kStatus_Success` – Message received successfully.
- `kStatus_InvalidArgument` – Invalid `readValue` pointer.

- `kStatus_Timeout` – Timeout occurred while waiting for RX register to be full.

Returns

`status_t`

`uint32_t MU_ReceiveMsg(MU_Type *base, uint32_t regIndex)`

Blocks to receive a message (infinite wait, no timeout protection).

This function waits until the RX register is full and receives the message. This function will wait indefinitely until a message is received.

Note: Both `MU_ReceiveMsg()` and `MU_ReceiveMsgTimeout()` are blocking functions. The difference is that `MU_ReceiveMsgTimeout()` includes timeout protection while this function waits indefinitely.

Warning: This function does not include timeout protection and may cause system hangs if the other core becomes unresponsive. For applications requiring timeout protection, use `MU_ReceiveMsgTimeout()` instead.

Parameters

- `base` – MU peripheral base address.
- `regIndex` – RX register index, see `mu_msg_reg_index_t`.

Returns

The received message.

`static inline void MU_SetFlagsNonBlocking(MU_Type *base, uint32_t flags)`

Sets the 3-bit MU flags reflect on the other MU side.

This function sets the 3-bit MU flags directly. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. The upper layer should make sure the status flag `kMU_FlagsUpdatingFlag` is cleared before calling this function.

```
while (kMU_FlagsUpdatingFlag & MU_GetStatusFlags(base))
{
    } Wait for previous MU flags updating.
```

```
MU_SetFlagsNonBlocking(base, 0U); Set the mU flags.
```

Parameters

- `base` – MU peripheral base address.
- `flags` – The 3-bit MU flags to set.

`status_t MU_SetFlags(MU_Type *base, uint32_t flags)`

Blocks setting the 3-bit MU flags reflect on the other MU side.

This function blocks setting the 3-bit MU flags. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. This function waits for the MU status flag `kMU_FlagsUpdatingFlag` cleared and sets the 3-bit MU flags.

If MU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

brief Blocks setting the 3-bit MU flags reflect on the other MU side.

This function blocks setting the 3-bit MU flags. Every time the 3-bit MU flags are changed, the status flag kMU_FlagsUpdatingFlag asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag kMU_FlagsUpdatingFlag is cleared by hardware. During the flags updating period, the flags cannot be changed. This function waits for the MU status flag kMU_FlagsUpdatingFlag cleared and sets the 3-bit MU flags.

If MU1_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

return status_t retval kStatus_Success Flags were set successfully. retval kStatus_Timeout Timeout occurred while waiting for flags to update.

Parameters

- base – MU peripheral base address.
- flags – The 3-bit MU flags to set.
- base – MU peripheral base address.
- flags – The 3-bit MU flags to set.

Return values

- kStatus_Success – Flags were set successfully.
- kStatus_Timeout – Timeout occurred while waiting for flags to update.

Returns

status_t

static inline uint32_t MU_GetFlags(MU_Type *base)

Gets the current value of the 3-bit MU flags set by the other side.

This function gets the current 3-bit MU flags on the current side.

Parameters

- base – MU peripheral base address.

Returns

flags Current value of the 3-bit flags.

static inline uint32_t MU_GetStatusFlags(MU_Type *base)

Gets the MU status flags.

This function returns the bit mask of the MU status flags. See _mu_status_flags.

```
uint32_t flags;
flags = MU_GetStatusFlags(base); Get all status flags.
if (kMU_Tx0EmptyFlag & flags)
{
    The TX0 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG0_VAL);
}
if (kMU_Tx1EmptyFlag & flags)
{
    The TX1 register is empty. Message can be sent.
```

(continues on next page)

(continued from previous page)

```

    MU_SendMsgNonBlocking(base, kMU_MsgReg1, MSG1_VAL);
}

```

This function returns the bit mask of the MU status flags. See `_mu_status_flags`.

```

uint32_t flags;
flags = MU_GetStatusFlags(base); Get all status flags.
if (kMU_Tx0EmptyFlag & flags)
{
    The TX0 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG0_VAL);
}
if (kMU_Tx1EmptyFlag & flags)
{
    The TX1 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg1, MSG1_VAL);
}

```

If there are more than 4 general purpose interrupts, use `MU_GetGeneralPurposeStatusFlags`.

Parameters

- base – MU peripheral base address.
- base – MU peripheral base address.

Returns

Bit mask of the MU status flags, see `_mu_status_flags`.

Returns

Bit mask of the MU status flags, see `_mu_status_flags`.

static inline uint32_t MU_GetRxStatusFlags(MU_Type *base)

Return the RX status flags.

This function return the RX status flags. Note: RFn bits of SR[27-24](mu status register) are mapped in reverse numerical order: RF0 -> SR[27] RF1 -> SR[26] RF2 -> SR[25] RF3 -> SR[24]

```
status_reg = MU_GetRxStatusFlags(base);
```

Parameters

- base – MU peripheral base address.

Returns

MU RX status

static inline uint32_t MU_GetInterruptsPending(MU_Type *base)

Gets the MU IRQ pending status of enabled interrupts.

This function returns the bit mask of the pending MU IRQs of enabled interrupts. Only these flags are checked. kMU_Tx0EmptyFlag kMU_Tx1EmptyFlag kMU_Tx2EmptyFlag kMU_Tx3EmptyFlag kMU_Rx0FullFlag kMU_Rx1FullFlag kMU_Rx2FullFlag kMU_Rx3FullFlag kMU_GenInt0Flag kMU_GenInt1Flag kMU_GenInt2Flag kMU_GenInt3Flag

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU IRQs pending.

```
static inline void MU_ClearStatusFlags(MU_Type *base, uint32_t mask)
```

Clears the specific MU status flags.

This function clears the specific MU status flags. The flags to clear should be passed in as bit mask. See `_mu_status_flags`.

Clear general interrupt 0 and general interrupt 1 pending flags.
`MU_ClearStatusFlags(base, kMU_GenInt0Flag | kMU_GenInt1Flag);`

Parameters

- base – MU peripheral base address.
- mask – Bit mask of the MU status flags. See `_mu_status_flags`. The following flags are cleared by hardware, this function could not clear them.
 - kMU_Tx0EmptyFlag
 - kMU_Tx1EmptyFlag
 - kMU_Tx2EmptyFlag
 - kMU_Tx3EmptyFlag
 - kMU_Rx0FullFlag
 - kMU_Rx1FullFlag
 - kMU_Rx2FullFlag
 - kMU_Rx3FullFlag
 - kMU_EventPendingFlag
 - kMU_FlagsUpdatingFlag
 - kMU_OtherSideInResetFlag

```
static inline void MU_EnableInterrupts(MU_Type *base, uint32_t mask)
```

Enables the specific MU interrupts.

This function enables the specific MU interrupts. The interrupts to enable should be passed in as bit mask. See `_mu_interrupt_enable`.

Enable general interrupt 0 and TX0 empty interrupt.
`MU_EnableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

Parameters

- base – MU peripheral base address.
- mask – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

```
static inline void MU_DisableInterrupts(MU_Type *base, uint32_t mask)
```

Disables the specific MU interrupts.

This function disables the specific MU interrupts. The interrupts to disable should be passed in as bit mask. See `_mu_interrupt_enable`.

Disable general interrupt 0 and TX0 empty interrupt.
`MU_DisableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

Parameters

- base – MU peripheral base address.
- mask – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

status_t MU_TriggerInterrupts(MU_Type *base, uint32_t mask)

Triggers interrupts to the other core.

This function triggers the specific interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `_mu_interrupt_trigger`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
if (kStatus_Success != MU_TriggerInterrupts(base, kMU_GenInt0InterruptTrigger | kMU_
↪GenInt2InterruptTrigger))
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

This function triggers the specific interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `_mu_interrupt_trigger`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
if (kStatus_Success != MU_TriggerInterrupts(base, kMU_GenInt0InterruptTrigger | kMU_
↪GenInt2InterruptTrigger))
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

If there are more than 4 general purpose interrupts, use `MU_TriggerGeneralPurposeInterrupts`.

Parameters

- base – MU peripheral base address.
- mask – Bit mask of the interrupts to trigger. See `_mu_interrupt_trigger`.
- base – MU peripheral base address.
- interrupts – Bit mask of the interrupts to trigger. See `_mu_interrupt_trigger`.

Return values

- kStatus_Success – Interrupts have been triggered successfully.
- kStatus_Fail – Previous interrupts have not been accepted.
- kStatus_Success – Interrupts have been triggered successfully.
- kStatus_Fail – Previous interrupts have not been accepted.

void MU_BootCoreB(MU_Type *base, mu_core_boot_mode_t mode)

Boots the core at B side.

This function sets the B side core's boot configuration and releases the core from reset.

Note: Only MU side A can use this function.

Parameters

- base – MU peripheral base address.
- mode – Core B boot mode.

static inline void MU_HoldCoreBReset(MU_Type *base)

Holds the core reset of B side.

This function causes the core of B side to be held in reset following any reset event.

Note: Only A side could call this function.

Parameters

- base – MU peripheral base address.

void MU_BootOtherCore(MU_Type *base, mu_core_boot_mode_t mode)

Boots the other core.

This function boots the other core with a boot configuration.

Parameters

- base – MU peripheral base address.
- mode – The other core boot mode.

static inline void MU_HoldOtherCoreReset(MU_Type *base)

Holds the other core reset.

This function causes the other core to be held in reset following any reset event.

Parameters

- base – MU peripheral base address.

static inline *status_t* MU_ResetBothSides(MU_Type *base)

Resets the MU for both A side and B side.

This function resets the MU for both A side and B side. Before reset, it is recommended to interrupt processor B, because this function may affect the ongoing processor B programs.

If MU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations if waiting for the other side to come out of reset takes too long.

Note: For some platforms, only MU side A could use this function, check reference manual for details.

Parameters

- base – MU peripheral base address.

Return values

- kStatus_Success – The MU was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other side to come out of reset.

Returns

status_t

status_t MU_HardwareResetOtherCore(MU_Type *base, bool waitReset, bool holdReset, mu_core_boot_mode_t bootMode)

Hardware reset the other core.

This function resets the other core, the other core could mask the hardware reset by calling `MU_MaskHardwareReset`. The hardware reset mask feature is only available for some platforms. This function could be used together with `MU_BootOtherCore` to control the other core reset workflow.

If `MU_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout` if waiting for the other core to enter or exit reset takes too long.

Example 1: Reset the other core, and no hold reset

```
MU_HardwareResetOtherCore(MU_A, true, false, bootMode);
```

In this example, the core at MU side B will reset with the specified boot mode.

Example 2: Reset the other core and hold it, then boot the other core later. Here the other core enters reset, and the reset is hold

```
MU_HardwareResetOtherCore(MU_A, true, true, modeDontCare);
```

Current core boot the other core when necessary.

```
MU_BootOtherCore(MU_A, bootMode);
```

This function resets the other core, the other core could mask the hardware reset by calling `MU_MaskHardwareReset`. The hardware reset mask feature is only available for some platforms. This function could be used together with `MU_BootOtherCore` to control the other core reset workflow.

If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout` if waiting for the other core to enter or exit reset takes too long.

Example 1: Reset the other core, and no hold reset

```
MU_HardwareResetOtherCore(MU_A, true, false, bootMode);
```

In this example, the core at MU side B will reset with the specified boot mode.

Example 2: Reset the other core and hold it, then boot the other core later. Here the other core enters reset, and the reset is hold

```
MU_HardwareResetOtherCore(MU_A, true, true, modeDontCare);
```

Current core boot the other core when necessary.

```
MU_BootOtherCore(MU_A, bootMode);
```

Note: The feature `waitReset`, `holdReset`, and `bootMode` might be not supported for some platforms. `waitReset` is only available for platforms that `FSL_FEATURE_MU_NO_CORE_STATUS` not defined as 1 and `FSL_FEATURE_MU_HAS_RESET_ASSERT_INT` not defined as 0. `holdReset` is only available for platforms that `FSL_FEATURE_MU_HAS_RSTH` not defined as 0. `bootMode` is only available for platforms that `FSL_FEATURE_MU_HAS_BOOT` not defined as 0.

Parameters

- `base` – MU peripheral base address.
- `waitReset` – Wait the other core enters reset.

- true: Wait until the other core enters reset, if the other core has masked the hardware reset, then this function will be blocked.
- false: Don't wait the reset.
- holdReset – Hold the other core reset or not.
 - true: Hold the other core in reset, this function returns directly when the other core enters reset.
 - false: Don't hold the other core in reset, this function waits until the other core out of reset.
- bootMode – Boot mode of the other core, if holdReset is true, this parameter is useless.
- base – MU peripheral base address.
- waitReset – Wait the other core enters reset. Only work when there is CSSR0[RAIP].
 - true: Wait until the other core enters reset, if the other core has masked the hardware reset, then this function will be blocked.
 - false: Don't wait the reset.
- holdReset – Hold the other core reset or not. Only work when there is CCR0[RSTH].
 - true: Hold the other core in reset, this function returns directly when the other core enters reset.
 - false: Don't hold the other core in reset, this function waits until the other core out of reset.
- bootMode – Boot mode of the other core, if holdReset is true, this parameter is useless.

Return values

- kStatus_Success – The other core was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other core to enter or exit reset.
- kStatus_Success – The other core was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other core to enter or exit reset.

Returns

status_t

Returns

status_t

static inline mu_power_mode_t MU_GetOtherCorePowerMode(MU_Type *base)

Gets the power mode of the other core.

This function gets the power mode of the other core.

Parameters

- base – MU peripheral base address.

Returns

Power mode of the other core.

uint32_t MU_GetInstance(MU_Type *base)

Get the MU instance index.

Parameters

- base – MU peripheral base address.

Returns

MU instance index.

static inline void MU_EnableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)

Enables the MU general purpose interrupts.

This function enables the MU general purpose interrupts. The interrupts to enable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_EnableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to enable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↳GeneralPurposeInterrupt3);
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the MU general purpose interrupts, see `mu_general_purpose_interrupt_t`.

static inline void MU_DisableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)

Disables the MU general purpose interrupts.

This function disables the MU general purpose interrupts. The interrupts to disable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_DisableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to disable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↳GeneralPurposeInterrupt3);
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the MU general purpose interrupts. see `mu_general_purpose_interrupt_t`.

static inline uint32_t MU_GetGeneralPurposeStatusFlags(MU_Type *base)

Gets the MU general purpose interrupt status flags.

This function returns the bit mask of the MU general purpose interrupt status flags. `MU_GetStatusFlags` can only get general purpose interrupt status 0~3, this function can get all general purpose interrupts status.

This example shows to check whether general purpose interrupt 0 and 3 happened.

```
uint32_t flags;
flags = MU_GetGeneralPurposeStatusFlags(base);
if (kMU_GeneralPurposeInterrupt0 & flags)
{
}
if (kMU_GeneralPurposeInterrupt3 & flags)
```

(continues on next page)

(continued from previous page)

```
{
}
```

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU general purpose interrupt status flags.

```
static inline void MU_ClearGeneralPurposeStatusFlags(MU_Type *base, uint32_t flags)
```

Clear the MU general purpose interrupt status flags.

This function clears the specific MU general purpose interrupt status flags. The flags to clear should be passed in as bit mask. `mu_general_purpose_interrupt_t_mu_status_flags`.

Example to clear general purpose interrupt 0 and general interrupt 1 pending flags.

```
MU_ClearGeneralPurposeStatusFlags(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt1);
```

Parameters

- base – MU peripheral base address.
- flags – Bit mask of the MU general purpose interrupt status flags. See `mu_general_purpose_interrupt_t`.

```
status_t MU_TriggerGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)
```

Triggers general purpose interrupts to the other core.

This function triggers the specific general purpose interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `mu_general_purpose_interrupt_t`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
status_t status;
status = MU_TriggerGeneralPurposeInterrupts(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt2);

if (kStatus_Success != status)
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the interrupts to trigger. See `mu_general_purpose_interrupt_t`.

Return values

- kStatus_Success – Interrupts have been triggered successfully.
- kStatus_Fail – Previous interrupts have not been accepted.

FSL_MU_DRIVER_VERSION

MU driver version.

FSL_MU_DRIVER_VERSION

MU driver version.

enum __mu_status_flags

MU status flags.

Values:

enumerator kMU_Tx0EmptyFlag

TX0 empty.

enumerator kMU_Tx1EmptyFlag

TX1 empty.

enumerator kMU_Tx2EmptyFlag

TX2 empty.

enumerator kMU_Tx3EmptyFlag

TX3 empty.

enumerator kMU_Rx0FullFlag

RX0 full.

enumerator kMU_Rx1FullFlag

RX1 full.

enumerator kMU_Rx2FullFlag

RX2 full.

enumerator kMU_Rx3FullFlag

RX3 full.

enumerator kMU_GenInt0Flag

General purpose interrupt 0 pending.

enumerator kMU_GenInt1Flag

General purpose interrupt 1 pending.

enumerator kMU_GenInt2Flag

General purpose interrupt 2 pending.

enumerator kMU_GenInt3Flag

General purpose interrupt 3 pending.

enumerator kMU_EventPendingFlag

MU event pending.

enumerator kMU_FlagsUpdatingFlag

MU flags update is on-going.

enumerator kMU_ResetAssertInterruptFlag

The other core reset assert interrupt pending.

enumerator kMU_ResetDeassertInterruptFlag

The other core reset de-assert interrupt pending.

enumerator kMU_MuResetInterruptFlag

The other side initializes MU reset.

enumerator kMU_HardwareResetInterruptFlag

Current side has been hardware reset by the other side.

enum `_mu_interrupt_enable`

MU interrupt source to enable.

Values:

enumerator `kMU_Tx0EmptyInterruptEnable`

TX0 empty.

enumerator `kMU_Tx1EmptyInterruptEnable`

TX1 empty.

enumerator `kMU_Tx2EmptyInterruptEnable`

TX2 empty.

enumerator `kMU_Tx3EmptyInterruptEnable`

TX3 empty.

enumerator `kMU_Rx0FullInterruptEnable`

RX0 full.

enumerator `kMU_Rx1FullInterruptEnable`

RX1 full.

enumerator `kMU_Rx2FullInterruptEnable`

RX2 full.

enumerator `kMU_Rx3FullInterruptEnable`

RX3 full.

enumerator `kMU_GenInt0InterruptEnable`

General purpose interrupt 0.

enumerator `kMU_GenInt1InterruptEnable`

General purpose interrupt 1.

enumerator `kMU_GenInt2InterruptEnable`

General purpose interrupt 2.

enumerator `kMU_GenInt3InterruptEnable`

General purpose interrupt 3.

enumerator `kMU_ResetAssertInterruptEnable`

The other core reset assert interrupt.

enumerator `kMU_ResetDeassertInterruptEnable`

The other core reset de-assert interrupt.

enumerator `kMU_MuResetInterruptEnable`

The other side initializes MU reset. The interrupt is ORed with the general purpose interrupt 3. The general purpose interrupt 3 is issued when the other side set the MU reset and this interrupt is enabled.

enumerator `kMU_HardwareResetInterruptEnable`

Current side has been hardware reset by the other side.

enum `_mu_interrupt_trigger`

MU interrupt that could be triggered to the other core.

Values:

enumerator `kMU_GenInt0InterruptTrigger`

General purpose interrupt 0.

enumerator kMU_GenInt1InterruptTrigger

General purpose interrupt 1.

enumerator kMU_GenInt2InterruptTrigger

General purpose interrupt 2.

enumerator kMU_GenInt3InterruptTrigger

General purpose interrupt 3.

enum __mu_msg_reg_index

MU message register.

Values:

enumerator kMU_MsgReg0

enumerator kMU_MsgReg1

enumerator kMU_MsgReg2

enumerator kMU_MsgReg3

enum __mu_status_flags

MU status flags.

Values:

enumerator kMU_Tx0EmptyFlag

TX0 empty.

enumerator kMU_Tx1EmptyFlag

TX1 empty.

enumerator kMU_Tx2EmptyFlag

TX2 empty.

enumerator kMU_Tx3EmptyFlag

TX3 empty.

enumerator kMU_Rx0FullFlag

RX0 full.

enumerator kMU_Rx1FullFlag

RX1 full.

enumerator kMU_Rx2FullFlag

RX2 full.

enumerator kMU_Rx3FullFlag

RX3 full.

enumerator kMU_GenInt0Flag

General purpose interrupt 0 pending.

enumerator kMU_GenInt1Flag

General purpose interrupt 1 pending.

enumerator kMU_GenInt2Flag

General purpose interrupt 2 pending.

enumerator kMU_GenInt3Flag

General purpose interrupt 3 pending.

enumerator kMU_RxFullPendingFlag

Any RX full flag is pending.

enumerator kMU_TxEmptyPendingFlag

Any TX empty flag is pending.

enumerator kMU_GenIntPendingFlag

Any general interrupt flag is pending.

enumerator kMU_EventPendingFlag

MU event pending.

enumerator kMU_FlagsUpdatingFlag

MU flags update is on-going.

enumerator kMU_MuInResetFlag

MU of any side is in reset.

enumerator kMU_MuResetInterruptFlag

The other side initializes MU reset.

enum _mu_interrupt_enable

MU interrupt source to enable.

Values:

enumerator kMU_Tx0EmptyInterruptEnable

TX0 empty.

enumerator kMU_Tx1EmptyInterruptEnable

TX1 empty.

enumerator kMU_Tx2EmptyInterruptEnable

TX2 empty.

enumerator kMU_Tx3EmptyInterruptEnable

TX3 empty.

enumerator kMU_Rx0FullInterruptEnable

RX0 full.

enumerator kMU_Rx1FullInterruptEnable

RX1 full.

enumerator kMU_Rx2FullInterruptEnable

RX2 full.

enumerator kMU_Rx3FullInterruptEnable

RX3 full.

enumerator kMU_GenInt0InterruptEnable

General purpose interrupt 0.

enumerator kMU_GenInt1InterruptEnable

General purpose interrupt 1.

enumerator kMU_GenInt2InterruptEnable

General purpose interrupt 2.

enumerator kMU_GenInt3InterruptEnable

General purpose interrupt 3.

enumerator kMU_MuResetInterruptEnable

The other side initializes MU reset.

enum _mu_interrupt_trigger

MU interrupt that could be triggered to the other core.

Values:

enumerator kMU_GenInt0InterruptTrigger

General purpose interrupt 0.

enumerator kMU_GenInt1InterruptTrigger

General purpose interrupt 1.

enumerator kMU_GenInt2InterruptTrigger

General purpose interrupt 2.

enumerator kMU_GenInt3InterruptTrigger

General purpose interrupt 3.

enum _mu_msg_reg_index

MU message register index.

Values:

enumerator kMU_MsgReg0

Message register 0.

enumerator kMU_MsgReg1

Message register 1.

enumerator kMU_MsgReg2

Message register 2.

enumerator kMU_MsgReg3

Message register 3.

enum _mu_general_purpose_interrupt

MU general purpose interrupts.

Values:

enumerator kMU_GeneralPurposeInterrupt0

General purpose interrupt 0

enumerator kMU_GeneralPurposeInterrupt1

General purpose interrupt 1

enumerator kMU_GeneralPurposeInterrupt2

General purpose interrupt 2

enumerator kMU_GeneralPurposeInterrupt3

General purpose interrupt 3

typedef enum _mu_msg_reg_index mu_msg_reg_index_t

MU message register.

typedef enum _mu_msg_reg_index mu_msg_reg_index_t

MU message register index.

typedef enum _mu_general_purpose_interrupt mu_general_purpose_interrupt_t

MU general purpose interrupts.

MU_CR_NMI_MASK

MU_BUSY_POLL_COUNT

Maximum polling iterations for MU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the MU code before timing out and returning an error.

It applies to all waiting loops in MU driver, such as waiting for TX register to be empty or waiting for RX register to be full.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if a core becomes unresponsive.

MU_GET_CORE_FLAG(flags)

MU_GET_STAT_FLAG(flags)

MU_GET_TX_FLAG(flags)

MU_GET_RX_FLAG(flags)

MU_GET_GI_FLAG(flags)

MU_CORE_INTR(intr)

MU_MISC_INTR(intr)

MU_TX_INTR(intr)

MU_RX_INTR(intr)

MU_GI_INTR(intr)

MU_GET_CORE_INTR(intrs)

MU_GET_TX_INTR(intrs)

MU_GET_RX_INTR(intrs)

MU_GET_GI_INTR(intrs)

MU_CORE_FLAG(flag)

MU_STAT_FLAG(flag)

MU_TX_FLAG(flag)

MU_RX_FLAG(flag)

MU_GI_FLAG(flag)

MU_GET_CORE_FLAG(flags)

MU_GET_STAT_FLAG(flags)

MU_GET_TX_FLAG(flags)

MU_GET_RX_FLAG(flags)

MU_GET_GI_FLAG(flags)

MU1_BUSY_POLL_COUNT

Maximum polling iterations for MU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the MU code before timing out and returning an error.

It applies to all waiting loops in MU driver, such as waiting for TX register to be empty or waiting for RX register to be full.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if a core becomes unresponsive.

2.56 PDM: Microphone Interface

2.57 PDM Driver

`void PDM_Init(PDM_Type *base, const pdm_config_t *config)`

Initializes the PDM peripheral.

Ungates the PDM clock, resets the module, and configures PDM with a configuration structure. The configuration structure can be custom filled or set with default values by `PDM_GetDefaultConfig()`.

Note: This API should be called at the beginning of the application to use the PDM driver. Otherwise, accessing the PDM module can cause a hard fault because the clock is not enabled.

Parameters

- `base` – PDM base pointer
- `config` – PDM configuration structure.

`void PDM_Deinit(PDM_Type *base)`

De-initializes the PDM peripheral.

This API gates the PDM clock. The PDM module can't operate unless `PDM_Init` is called to enable the clock.

Parameters

- `base` – PDM base pointer

`static inline void PDM_Reset(PDM_Type *base)`

Resets the PDM module.

Parameters

- `base` – PDM base pointer

`static inline void PDM_Enable(PDM_Type *base, bool enable)`

Enables/disables PDM interface.

Parameters

- `base` – PDM base pointer
- `enable` – True means PDM interface is enabled, false means PDM interface is disabled.

```
static inline void PDM_EnableDebugMode(PDM_Type *base, bool enable)
```

Enables/disables debug mode for PDM. The PDM interface cannot enter debug mode once in Disable/Low Leakage or Low Power mode.

Parameters

- base – PDM base pointer
- enable – True means PDM interface enter debug mode, false means PDM interface in normal mode.

```
static inline void PDM_EnableInDebugMode(PDM_Type *base, bool enable)
```

Enables/disables PDM interface in debug mode.

Parameters

- base – PDM base pointer
- enable – True means PDM interface is enabled debug mode, false means PDM interface is disabled after after completing the current frame in debug mode.

```
static inline void PDM_EnterLowLeakageMode(PDM_Type *base, bool enable)
```

Enables/disables PDM interface disable/Low Leakage mode.

Parameters

- base – PDM base pointer
- enable – True means PDM interface is in disable/low leakage mode, False means PDM interface is in normal mode.

```
static inline void PDM_EnableChannel(PDM_Type *base, uint8_t channel, bool enable)
```

Enables/disables the PDM channel.

Parameters

- base – PDM base pointer
- channel – PDM channel number need to enable or disable.
- enable – True means enable PDM channel, false means disable.

```
void PDM_SetChannelConfig(PDM_Type *base, uint32_t channel, const pdm_channel_config_t *config)
```

PDM one channel configurations.

Parameters

- base – PDM base pointer
- config – PDM channel configurations.
- channel – channel number. after completing the current frame in debug mode.

```
status_t PDM_SetSampleRateConfig(PDM_Type *base, uint32_t sourceClock_HZ, uint32_t sampleRate_HZ)
```

PDM set sample rate.

Note: This function is depend on the configuration of the PDM and PDM channel, so the correct call sequence is

```
PDM_Init(base, pdmConfig)
PDM_SetChannelConfig(base, channel, &channelConfig)
PDM_SetSampleRateConfig(base, source, sampleRate)
```

Parameters

- base – PDM base pointer
- sourceClock_HZ – PDM source clock frequency.
- sampleRate_HZ – PDM sample rate.

status_t PDM_SetSampleRate(PDM_Type *base, uint32_t enableChannelMask,
pdm_df_quality_mode_t qualityMode, uint8_t osr, uint32_t clkDiv)

PDM set sample rate.

Deprecated:

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

Parameters

- base – PDM base pointer
- enableChannelMask – PDM channel enable mask.
- qualityMode – quality mode.
- osr – cic oversample rate
- clkDiv – clock divider

uint32_t PDM_GetInstance(PDM_Type *base)

Get the instance number for PDM.

Parameters

- base – PDM base pointer.

static inline uint32_t PDM_GetStatus(PDM_Type *base)

Gets the PDM internal status flag. Use the Status Mask in `_pdm_internal_status` to get the status value needed.

Parameters

- base – PDM base pointer

Returns

PDM status flag value.

static inline uint32_t PDM_GetFifoStatus(PDM_Type *base)

Gets the PDM FIFO status flag. Use the Status Mask in `_pdm_fifo_status` to get the status value needed.

Parameters

- base – PDM base pointer

Returns

FIFO status.

static inline uint32_t PDM_GetRangeStatus(PDM_Type *base)

Gets the PDM Range status flag. Use the Status Mask in `_pdm_range_status` to get the status value needed.

Parameters

- base – PDM base pointer

Returns

output status.

static inline void PDM_ClearStatus(PDM_Type *base, uint32_t mask)

Clears the PDM Tx status.

Parameters

- base – PDM base pointer
- mask – State mask. It can be a combination of the status between kPDM_StatusFrequencyLow and kPDM_StatusCh7FifoDataAvaliable.

static inline void PDM_ClearFIFOStatus(PDM_Type *base, uint32_t mask)

Clears the PDM Tx status.

Parameters

- base – PDM base pointer
- mask – State mask. It can be a combination of the status in _pdm_fifo_status.

static inline void PDM_ClearRangeStatus(PDM_Type *base, uint32_t mask)

Clears the PDM range status.

Parameters

- base – PDM base pointer
- mask – State mask. It can be a combination of the status in _pdm_range_status.

void PDM_EnableInterrupts(PDM_Type *base, uint32_t mask)

Enables the PDM interrupt requests.

Parameters

- base – PDM base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kPDM_ErrorInterruptEnable
 - kPDM_FIFOInterruptEnable

static inline void PDM_DisableInterrupts(PDM_Type *base, uint32_t mask)

Disables the PDM interrupt requests.

Parameters

- base – PDM base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kPDM_ErrorInterruptEnable
 - kPDM_FIFOInterruptEnable

static inline void PDM_EnableDMA(PDM_Type *base, bool enable)

Enables/disables the PDM DMA requests.

Parameters

- base – PDM base pointer
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t PDM_GetDataRegisterAddress(PDM_Type *base, uint32_t channel)

Gets the PDM data register address.

This API is used to provide a transfer address for the PDM DMA transfer configuration.

Parameters

- base – PDM base pointer.
- channel – Which data channel used.

Returns

data register address.

```
void PDM_ReadFifo(PDM_Type *base, uint32_t startChannel, uint32_t channelNums, void  
                 *buffer, size_t size, uint32_t dataWidth)
```

PDM read fifo.

Note: : This function support 16 bit only for IP version that only supports 16bit.

Parameters

- base – PDM base pointer.
- startChannel – start channel number.
- channelNums – total enabled channelnums.
- buffer – received buffer address.
- size – number of samples to read.
- dataWidth – sample width.

```
void PDM_SetChannelGain(PDM_Type *base, uint32_t channel, pdm_df_output_gain_t gain)
```

Set the PDM channel gain.

Please note for different quality mode, the valid gain value is different, reference RM for detail.

Parameters

- base – PDM base pointer.
- channel – PDM channel index.
- gain – channel gain, the register gain value range is 0 - 15.

```
void PDM_SetHwvadConfig(PDM_Type *base, const pdm_hwvad_config_t *config)
```

Configure voice activity detector.

Parameters

- base – PDM base pointer
- config – Voice activity detector configure structure pointer .

```
static inline void PDM_ForceHwvadOutputDisable(PDM_Type *base, bool enable)
```

PDM hwvad force output disable.

Parameters

- base – PDM base pointer
- enable – true is output force disable, false is output not force.

```
static inline void PDM_ResetHwvad(PDM_Type *base)
```

PDM hwvad reset. It will reset VADNDATA register and will clean all internal buffers, should be called when the PDM isn't running.

Parameters

- base – PDM base pointer

static inline void PDM_EnableHwvad(PDM_Type *base, bool enable)

Enable/Disable Voice activity detector. Should be called when the PDM isn't running.

Parameters

- base – PDM base pointer.
- enable – True means enable voice activity detector, false means disable.

static inline void PDM_EnableHwvadInterrupts(PDM_Type *base, uint32_t mask)

Enables the PDM Voice Detector interrupt requests.

Parameters

- base – PDM base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kPDM_HWVADErrorInterruptEnable
 - kPDM_HWVADInterruptEnable

static inline void PDM_DisableHwvadInterrupts(PDM_Type *base, uint32_t mask)

Disables the PDM Voice Detector interrupt requests.

Parameters

- base – PDM base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kPDM_HWVADErrorInterruptEnable
 - kPDM_HWVADInterruptEnable

static inline void PDM_ClearHwvadInterruptStatusFlags(PDM_Type *base, uint32_t mask)

Clears the PDM voice activity detector status flags.

Parameters

- base – PDM base pointer
- mask – State mask,reference _pdm_hwvad_int_status.

static inline uint32_t PDM_GetHwvadInterruptStatusFlags(PDM_Type *base)

Clears the PDM voice activity detector status flags.

Parameters

- base – PDM base pointer

Returns

status, reference _pdm_hwvad_int_status

static inline uint32_t PDM_GetHwvadInitialFlag(PDM_Type *base)

Get the PDM voice activity detector initial flags.

Parameters

- base – PDM base pointer

Returns

initial flag.

static inline void PDM_EnableHwvadSignalFilter(PDM_Type *base, bool enable)

Enables/disables voice activity detector signal filter.

Parameters

- base – PDM base pointer
- enable – True means enable signal filter, false means disable.

```
void PDM_SetHwvadSignalFilterConfig(PDM_Type *base, bool enableMaxBlock, uint32_t  
                                   signalGain)
```

Configure voice activity detector signal filter.

Parameters

- base – PDM base pointer
- enableMaxBlock – If signal maximum block enabled.
- signalGain – Gain value for the signal energy.

```
void PDM_SetHwvadNoiseFilterConfig(PDM_Type *base, const pdm_hwvad_noise_filter_t *config)
```

Configure voice activity detector noise filter.

Parameters

- base – PDM base pointer
- config – Voice activity detector noise filter configure structure pointer .

```
static inline void PDM_EnableHwvadZeroCrossDetector(PDM_Type *base, bool enable)
```

Enables/disables voice activity detector zero cross detector.

Parameters

- base – PDM base pointer
- enable – True means enable zero cross detector, false means disable.

```
void PDM_SetHwvadZeroCrossDetectorConfig(PDM_Type *base, const  
                                          pdm_hwvad_zero_cross_detector_t *config)
```

Configure voice activity detector zero cross detector.

Parameters

- base – PDM base pointer
- config – Voice activity detector zero cross detector configure structure pointer .

```
static inline uint16_t PDM_GetNoiseData(PDM_Type *base)
```

Reads noise data.

Parameters

- base – PDM base pointer.

Returns

Data in PDM noise data register.

```
static inline void PDM_SetHwvadInternalFilterStatus(PDM_Type *base,  
                                                    pdm_hwvad_filter_status_t status)
```

set hwvad internal filter status . Note: filter initial status should be asserted for two more cycles, then set it to normal operation.

Parameters

- base – PDM base pointer.
- status – internal filter status.

```
void PDM_SetHwvadInEnvelopeBasedMode(PDM_Type *base, const pdm_hwvad_config_t  
                                     *hwvadConfig, const pdm_hwvad_noise_filter_t  
                                     *noiseConfig, const  
                                     pdm_hwvad_zero_cross_detector_t *zcdConfig,  
                                     uint32_t signalGain)
```


set HWVAD in envelope based mode . Recommend configurations,

```
static const pdm_hwvad_config_t hwvadConfig = {
    .channel      = 0,
    .initializeTime = 10U,
    .cicOverSampleRate = 0U,
    .inputGain     = 0U,
    .frameTime     = 10U,
    .cutOffFreq    = kPDM_HwvadHpfBypassed,
    .enableFrameEnergy = false,
    .enablePreFilter = true,
};

static const pdm_hwvad_noise_filter_t noiseFilterConfig = {
    .enableAutoNoiseFilter = false,
    .enableNoiseMin        = true,
    .enableNoiseDecimation = true,
    .noiseFilterAdjustment = 0U,
    .noiseGain              = 7U,
    .enableNoiseDetectOR    = true,
};
```

Parameters

- base – PDM base pointer.
- hwvadConfig – internal filter status.
- noiseConfig – Voice activity detector noise filter configure structure pointer.
- zcdConfig – Voice activity detector zero cross detector configure structure pointer .
- signalGain – signal gain value.

```
void PDM_SetHwvadInEnergyBasedMode(PDM_Type *base, const pdm_hwvad_config_t
                                   *hwvadConfig, const pdm_hwvad_noise_filter_t
                                   *noiseConfig, const pdm_hwvad_zero_cross_detector_t
                                   *zcdConfig, uint32_t signalGain)
```

brief set HWVAD in energy based mode . Recommend configurations, code static const pdm_hwvad_config_t hwvadConfig = { .channel = 0, .initializeTime = 10U, .cicOverSampleRate = 0U, .inputGain = 0U, .frameTime = 10U, .cutOffFreq = kPDM_HwvadHpfBypassed, .enableFrameEnergy = true, .enablePreFilter = true, };

static const pdm_hwvad_noise_filter_t noiseFilterConfig = { .enableAutoNoiseFilter = true, .enableNoiseMin = false, .enableNoiseDecimation = false, .noiseFilterAdjustment = 0U, .noiseGain = 7U, .enableNoiseDetectOR = false, }; code param base PDM base pointer. param hwvadConfig internal filter status. param noiseConfig Voice activity detector noise filter configure structure pointer. param zcdConfig Voice activity detector zero cross detector configure structure pointer . param signalGain signal gain value, signal gain value should be properly according to application.

```
void PDM_EnableHwvadInterruptCallback(PDM_Type *base, pdm_hwvad_callback_t vadCallback,
                                       void *userData, bool enable)
```

Enable/Disable hwvad callback.

This function enable/disable the hwvad interrupt for the selected PDM peripheral.

Parameters

- base – Base address of the PDM peripheral.

- `vadCallback` – callback Pointer to store callback function, should be NULL when disable.
- `userData` – user data.
- `enable` – true is enable, false is disable.

Return values

None. –

```
void PDM_TransferCreateHandle(PDM_Type *base, pdm_handle_t *handle,  
                             pdm_transfer_callback_t callback, void *userData)
```

Initializes the PDM handle.

This function initializes the handle for the PDM transactional APIs. Call this function once to get the handle initialized.

Parameters

- `base` – PDM base pointer.
- `handle` – PDM handle pointer.
- `callback` – Pointer to the user callback function.
- `userData` – User parameter passed to the callback function.

```
status_t PDM_TransferSetChannelConfig(PDM_Type *base, pdm_handle_t *handle, uint32_t  
                                       channel, const pdm_channel_config_t *config, uint32_t  
                                       format)
```

PDM set channel transfer config.

Parameters

- `base` – PDM base pointer.
- `handle` – PDM handle pointer.
- `channel` – PDM channel.
- `config` – channel config.
- `format` – data format, support data width configurations, `_pdm_data_width`.

Return values

`kStatus_PDM_ChannelConfig_Failed` – or `kStatus_Success`.

```
status_t PDM_TransferReceiveNonBlocking(PDM_Type *base, pdm_handle_t *handle,  
                                         pdm_transfer_t *xfer)
```

Performs an interrupt non-blocking receive transfer on PDM.

Note: This API returns immediately after the transfer initiates. Call the `PDM_RxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_PDM_Busy`, the transfer is finished.

Parameters

- `base` – PDM base pointer
- `handle` – Pointer to the `pdm_handle_t` structure which stores the transfer state.
- `xfer` – Pointer to the `pdm_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully started the data receive.

- kStatus_PDM_Busy – Previous receive still not finished.

void PDM_TransferAbortReceive(PDM_Type *base, *pdm_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 – PDM base pointer
- handle – Pointer to the *pdm_handle_t* structure which stores the transfer state.

void PDM_TransferHandleIRQ(PDM_Type *base, *pdm_handle_t* *handle)

Tx interrupt handler.

Parameters

- base – PDM base pointer.
- handle – Pointer to the *pdm_handle_t* structure.

FSL_PDM_DRIVER_VERSION

Version 2.9.3

PDM return status.

Values:

enumerator kStatus_PDM_Busy

PDM is busy.

enumerator kStatus_PDM_CLK_LOW

PDM clock frequency low

enumerator kStatus_PDM_FIFO_ERROR

PDM FIFO underrun or overflow

enumerator kStatus_PDM_QueueFull

PDM FIFO underrun or overflow

enumerator kStatus_PDM_Idle

PDM is idle

enumerator kStatus_PDM_Output_ERROR

PDM is output error

enumerator kStatus_PDM_ChannelConfig_Failed

PDM channel config failed

enumerator kStatus_PDM_HWVAD_VoiceDetected

PDM hwvad voice detected

enumerator kStatus_PDM_HWVAD_Error

PDM hwvad error

enum __pdm_interrupt_enable

The PDM interrupt enable flag.

Values:

enumerator kPDM_ErrorInterruptEnable
PDM channel error interrupt enable.

enumerator kPDM_FIFOInterruptEnable
PDM channel FIFO interrupt

enum __pdm_internal_status
The PDM status.

Values:

enumerator kPDM_StatusDfBusyFlag
Decimation filter is busy processing data

enumerator kPDM_StatusFrequencyLow
Mic app clock frequency not high enough

enumerator kPDM_StatusCh0FifoDataAvaliable
channel 0 fifo data reached watermark level

enumerator kPDM_StatusCh1FifoDataAvaliable
channel 1 fifo data reached watermark level

enumerator kPDM_StatusCh2FifoDataAvaliable
channel 2 fifo data reached watermark level

enumerator kPDM_StatusCh3FifoDataAvaliable
channel 3 fifo data reached watermark level

enum __pdm_channel_enable_mask
PDM channel enable mask.

Values:

enumerator kPDM_EnableChannel0
channgel 0 enable mask

enumerator kPDM_EnableChannel1
channgel 1 enable mask

enumerator kPDM_EnableChannel2
channgel 2 enable mask

enumerator kPDM_EnableChannel3
channgel 3 enable mask

enumerator kPDM_EnableChannelAll

enum __pdm_fifo_status
The PDM fifo status.

Values:

enumerator kPDM_FifoStatusUnderflowCh0
channel0 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh1
channel1 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh2
channel2 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh3
channel3 fifo status underflow

enumerator kPDM_FifoStatusOverflowCh0
channel0 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh1
channel1 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh2
channel2 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh3
channel3 fifo status overflow

enum __pdm_range_status
The PDM output status.

Values:

enumerator kPDM_RangeStatusUnderFlowCh0
channel0 range status underflow

enumerator kPDM_RangeStatusUnderFlowCh1
channel1 range status underflow

enumerator kPDM_RangeStatusUnderFlowCh2
channel2 range status underflow

enumerator kPDM_RangeStatusUnderFlowCh3
channel3 range status underflow

enumerator kPDM_RangeStatusOverFlowCh0
channel0 range status overflow

enumerator kPDM_RangeStatusOverFlowCh1
channel1 range status overflow

enumerator kPDM_RangeStatusOverFlowCh2
channel2 range status overflow

enumerator kPDM_RangeStatusOverFlowCh3
channel3 range status overflow

enum __pdm_dc_removal
PDM DC removal configurations.

Values:

enumerator kPDM_DcRemovalCutOff20Hz
DC removal cut off 20HZ

enumerator kPDM_DcRemovalCutOff13Hz
DC removal cut off 13.3HZ

enumerator kPDM_DcRemovalCutOff40Hz
DC removal cut off 40HZ

enumerator kPDM_DcRemovalBypass
DC removal bypass

enum __pdm_df_quality_mode
PDM decimation filter quality mode.

Values:

enumerator kPDM_QualityModeMedium
quality mode memdium

enumerator kPDM_QualityModeHigh
quality mode high

enumerator kPDM_QualityModeLow
quality mode low

enumerator kPDM_QualityModeVeryLow0
quality mode very low0

enumerator kPDM_QualityModeVeryLow1
quality mode very low1

enumerator kPDM_QualityModeVeryLow2
quality mode very low2

enum __pdm_qulaity__mode_k_factor
PDM quality mode K factor.

Values:

enumerator kPDM_QualityModeHighKFactor
high quality mode K factor = 1 / 2

enumerator kPDM_QualityModeMediumKFactor
medium/very low0 quality mode K factor = 2 / 2

enumerator kPDM_QualityModeLowKFactor
low/very low1 quality mode K factor = 4 / 2

enumerator kPDM_QualityModeVeryLow2KFactor
very low2 quality mode K factor = 8 / 2

enum __pdm_df_output_gain
PDM decimation filter output gain.

Values:

enumerator kPDM_DfOutputGain0
Decimation filter output gain 0

enumerator kPDM_DfOutputGain1
Decimation filter output gain 1

enumerator kPDM_DfOutputGain2
Decimation filter output gain 2

enumerator kPDM_DfOutputGain3
Decimation filter output gain 3

enumerator kPDM_DfOutputGain4
Decimation filter output gain 4

enumerator kPDM_DfOutputGain5
Decimation filter output gain 5

enumerator kPDM_DfOutputGain6
Decimation filter output gain 6

enumerator kPDM_DfOutputGain7
Decimation filter output gain 7

enumerator kPDM_DfOutputGain8
Decimation filter output gain 8

enumerator kPDM_DfOutputGain9
Decimation filter output gain 9

enumerator kPDM_DfOutputGain10
Decimation filter output gain 10

enumerator kPDM_DfOutputGain11
Decimation filter output gain 11

enumerator kPDM_DfOutputGain12
Decimation filter output gain 12

enumerator kPDM_DfOutputGain13
Decimation filter output gain 13

enumerator kPDM_DfOutputGain14
Decimation filter output gain 14

enumerator kPDM_DfOutputGain15
Decimation filter output gain 15

enum __pdm_data_width
PDM data width.
Values:

enumerator kPDM_DataWwidth24
PDM data width 24bit

enumerator kPDM_DataWwidth32
PDM data width 32bit

enum __pdm_hwvad_interrupt_enable
PDM voice activity detector interrupt type.
Values:

enumerator kPDM_HwvadErrorInterruptEnable
PDM channel HWVAD error interrupt enable.

enumerator kPDM_HwvadInterruptEnable
PDM channel HWVAD interrupt

enum __pdm_hwvad_int_status
The PDM hwvad interrupt status flag.
Values:

enumerator kPDM_HwvadStatusInputSaturation
HWVAD saturation condition

enumerator kPDM_HwvadStatusVoiceDetectFlag
HWVAD voice detect interrupt triggered

enum __pdm_hwvad_hpf_config
High pass filter configure cut-off frequency.
Values:

enumerator kPDM_HwvadHpfBypassed
High-pass filter bypass

enumerator kPDM_HwvadHpfCutOffFreq1750Hz
High-pass filter cut off frequency 1750HZ

enumerator kPDM_HwvadHpfCutOffFreq215Hz
High-pass filter cut off frequency 215HZ

enumerator kPDM_HwvadHpfCutOffFreq102Hz
High-pass filter cut off frequency 102HZ

enum __pdm_hwvad_filter_status
HWVAD internal filter status.

Values:

enumerator kPDM_HwvadInternalFilterNormalOperation
internal filter ready for normal operation

enumerator kPDM_HwvadInternalFilterInitial
interla filter are initial

enum __pdm_hwvad_zcd_result
PDM voice activity detector zero cross detector result.

Values:

enumerator kPDM_HwvadResultOREnergyBasedDetection
zero cross detector result will be OR with energy based detection

enumerator kPDM_HwvadResultANDEnergyBasedDetection
zero cross detector result will be AND with energy based detection

typedef enum __pdm_dc_removal pdm_dc_removal_t
PDM DC remover configurations.

typedef enum __pdm_df_quality_mode pdm_df_quality_mode_t
PDM decimation filter quality mode.

typedef enum __pdm_df_output_gain pdm_df_output_gain_t
PDM decimation filter output gain.

typedef struct __pdm_channel_config pdm_channel_config_t
PDM channel configurations.

typedef struct __pdm_config pdm_config_t
PDM user configuration structure.

typedef enum __pdm_hwvad_hpf_config pdm_hwvad_hpf_config_t
High pass filter configure cut-off frequency.

typedef enum __pdm_hwvad_filter_status pdm_hwvad_filter_status_t
HWVAD internal filter status.

typedef struct __pdm_hwvad_config pdm_hwvad_config_t
PDM voice activity detector user configuration structure.

typedef struct __pdm_hwvad_noise_filter pdm_hwvad_noise_filter_t
PDM voice activity detector noise filter user configuration structure.

typedef enum __pdm_hwvad_zcd_result pdm_hwvad_zcd_result_t
PDM voice activity detector zero cross detector result.

typedef struct __pdm_hwvad_zero_cross_detector pdm_hwvad_zero_cross_detector_t
PDM voice activity detector zero cross detector configuration structure.


```
typedef struct _pdm_transfer pdm_transfer_t
```

PDM SDMA transfer structure.

```
typedef struct _pdm_handle pdm_handle_t
```

PDM handle.

```
typedef void (*pdm_transfer_callback_t)(PDM_Type *base, pdm_handle_t *handle, status_t status, void *userData)
```

PDM transfer callback prototype.

```
typedef void (*pdm_hwvad_callback_t)(status_t status, void *userData)
```

PDM HWVAD callback prototype.

```
typedef struct _pdm_hwvad_notification pdm_hwvad_notification_t
```

PDM HWVAD notification structure.

```
PDM_XFER_QUEUE_SIZE
```

PDM XFER QUEUE SIZE.

```
struct __pdm_channel_config
```

#include <fsl_pdm.h> PDM channel configurations.

Public Members

pdm_dc_removal_t outputCutOffFreq

PDM output DC remover cut off frequency

pdm_df_output_gain_t gain

Decimation Filter Output Gain

```
struct __pdm_config
```

#include <fsl_pdm.h> PDM user configuration structure.

Public Members

bool enableDoze

This module will enter disable/low leakage mode if DOZEN is active with ipg_doze is asserted

bool enableFilterBypass

Switchable bypass path for the decimation filter

uint8_t fifoWatermark

Watermark value for FIFO

pdm_df_quality_mode_t qualityMode

Quality mode

uint8_t cicOverSampleRate

CIC filter over sampling rate

```
struct __pdm_hwvad_config
```

#include <fsl_pdm.h> PDM voice activity detector user configuration structure.

Public Members

uint8_t channel

Which channel uses voice activity detector

`uint8_t initializeTime`
Number of frames or samples to initialize voice activity detector.

`uint8_t cicOverSampleRate`
CIC filter over sampling rate

`uint8_t inputGain`
Voice activity detector input gain

`uint32_t frameTime`
Voice activity frame time

`pdm_hwvad_hpf_config_t cutOffFreq`
High pass filter cut off frequency

`bool enableFrameEnergy`
If frame energy enabled, true means enable

`bool enablePreFilter`
If pre-filter enabled

`struct __pdm_hwvad_noise_filter`
#include <fsl_pdm.h> PDM voice activity detector noise filter user configuration structure.

Public Members

`bool enableAutoNoiseFilter`
If noise filter automatically activated, true means enable

`bool enableNoiseMin`
If Noise minimum block enabled, true means enabled

`bool enableNoiseDecimation`
If enable noise input decimation

`bool enableNoiseDetectOR`
Enables a OR logic in the output of minimum noise estimator block

`uint32_t noiseFilterAdjustment`
The adjustment value of the noise filter

`uint32_t noiseGain`
Gain value for the noise energy or envelope estimated

`struct __pdm_hwvad_zero_cross_detector`
#include <fsl_pdm.h> PDM voice activity detector zero cross detector configuration structure.

Public Members

`bool enableAutoThreshold`
If ZCD auto-threshold enabled, true means enabled.

`pdm_hwvad_zcd_result_t zcdAnd`
Is ZCD result is AND'ed with energy-based detection, false means OR'ed

`uint32_t threshold`
The adjustment value of the noise filter

uint32_t adjustmentThreshold

Gain value for the noise energy or envelope estimated

struct __pdm_transfer

#include <fsl_pdm.h> PDM SDMA transfer structure.

Public Members

volatile uint8_t *data

Data start address to transfer.

volatile size_t dataSize

Total Transfer bytes size.

struct __pdm_hwvad_notification

#include <fsl_pdm.h> PDM HWVAD notification structure.

struct __pdm_handle

#include <fsl_pdm.h> PDM handle structure.

Public Members

uint32_t state

Transfer status

pdm_transfer_callback_t callback

Callback function called at transfer event

void *userData

Callback parameter passed to callback function

pdm_transfer_t pdmQueue[(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

uint32_t format

data format

uint8_t watermark

Watermark value

uint8_t startChannel

end channel

uint8_t channelNums

Enabled channel number

2.58 PDM EDMA Driver

```
void PDM_TransferInstallEDMATCDMemory(pdm_edma_handle_t *handle, void *tcdAddr, size_t  
tcdNum)
```

Install EDMA descriptor memory.

Parameters

- handle – Pointer to EDMA channel transfer handle.
- tcdAddr – EDMA head descriptor address.
- tcdNum – EDMA link descriptor address.

```
void PDM_TransferCreateHandleEDMA(PDM_Type *base, pdm_edma_handle_t *handle,  
pdm_edma_callback_t callback, void *userData,  
edma_handle_t *dmaHandle)
```

Initializes the PDM Rx eDMA handle.

This function initializes the PDM slave DMA handle, which can be used for other PDM master transactional APIs. Usually, for a specified PDM instance, call this API once to get the initialized handle.

Parameters

- base – PDM base pointer.
- handle – PDM eDMA handle pointer.
- callback – Pointer to user callback function.
- userData – User parameter passed to the callback function.
- dmaHandle – eDMA handle pointer, this handle shall be static allocated by users.

```
void PDM_TransferSetMultiChannelInterleaveType(pdm_edma_handle_t *handle,  
pdm_edma_multi_channel_interleave_t  
multiChannelInterleaveType)
```

Initializes the multi PDM channel interleave type.

This function initializes the PDM DMA handle member `interleaveType`, it shall be called only when application would like to use type `kPDM_EDMAMultiChannelInterleavePerChannelBlock`, since the default `interleaveType` is `kPDM_EDMAMultiChannelInterleavePerChannelSample` always

Parameters

- handle – PDM eDMA handle pointer.
- multiChannelInterleaveType – Multi channel interleave type.

```
void PDM_TransferSetChannelConfigEDMA(PDM_Type *base, pdm_edma_handle_t *handle,  
uint32_t channel, const pdm_channel_config_t  
*config)
```

Configures the PDM channel.

Parameters

- base – PDM base pointer.
- handle – PDM eDMA handle pointer.
- channel – channel index.
- config – pdm channel configurations.

```
status_t PDM_TransferReceiveEDMA(PDM_Type *base, pdm_edma_handle_t *handle,
                                pdm_edma_transfer_t *xfer)
```

Performs a non-blocking PDM receive using eDMA.

Mcaro MCUX_SDK_PDM_EDMA_PDM_ENABLE_INTERNAL can control whether PDM is enabled internally or externally.

- a. Scatter gather case: This funcio support dynamic scatter gather and staic scatter gather, a. for the dynamic scatter gather case: Application should call PDM_TransferReceiveEDMA function continuously to make sure new receive request is submit before the previous one finish. b. for the static scatter gather case: Application should use the link transfer feature and make sure a loop link transfer is provided, such as:

```
pdm_edma_transfer_t pdmXfer[2] =
{
    {
        .data = s_buffer,
        .dataSize = BUFFER_SIZE,
        .linkTransfer = &pdmXfer[1],
    },
    {
        .data = &s_buffer[BUFFER_SIZE],
        .dataSize = BUFFER_SIZE,
        .linkTransfer = &pdmXfer[0]
    },
};
```

- b. Multi channel case: This function support receive multi pdm channel data, for example, if two channel is requested,

```
PDM_TransferSetChannelConfigEDMA(DEMO_PDM, &s_pdmRxHandle_0, DEMO_PDM_
↳ENABLE_CHANNEL_0, &channelConfig);
PDM_TransferSetChannelConfigEDMA(DEMO_PDM, &s_pdmRxHandle_0, DEMO_PDM_
↳ENABLE_CHANNEL_1, &channelConfig);
PDM_TransferReceiveEDMA(DEMO_PDM, &s_pdmRxHandle_0, pdmXfer);
```

The output data will be formatted as below if handle->interleaveType =

Note: This interface returns immediately after the transfer initiates. Call the PDM_GetReceiveRemainingBytes to poll the transfer status and check whether the PDM transfer is finished.

```
void PDM_TransferTerminateReceiveEDMA(PDM_Type *base, pdm_edma_handle_t *handle)
```

Terminate all PDM receive.

This function will clear all transfer slots buffered in the pdm queue. If users only want to abort the current transfer slot, please call PDM_TransferAbortReceiveEDMA.

Parameters

- base – PDM base pointer.
- handle – PDM eDMA handle pointer.

```
void PDM_TransferAbortReceiveEDMA(PDM_Type *base, pdm_edma_handle_t *handle)
```

Aborts a PDM 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 PDM_TransferTerminateReceiveEDMA.

Parameters

- base – PDM base pointer
- handle – PDM eDMA handle pointer.

status_t PDM_TransferGetReceiveCountEDMA(PDM_Type *base, *pdm_edma_handle_t* *handle, size_t *count)

Gets byte count received by PDM.

Parameters

- base – PDM base pointer
- handle – PDM eDMA handle pointer.
- count – Bytes count received by PDM.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is no non-blocking transaction in progress.

FSL_PDM_EDMA_DRIVER_VERSION

Version 2.6.5

enum __pdm_edma_multi_channel_interleave

pdm multi channel interleave type

Values:

enumerator kPDM_EDMAMultiChannelInterleavePerChannelSample

enumerator kPDM_EDMAMultiChannelInterleavePerChannelBlock

typedef struct *pdm_edma_handle* pdm_edma_handle_t

PDM edma handler.

typedef enum *pdm_edma_multi_channel_interleave* pdm_edma_multi_channel_interleave_t

pdm multi channel interleave type

typedef struct *pdm_edma_transfer* pdm_edma_transfer_t

PDM edma transfer.

typedef void (*pdm_edma_callback_t)(PDM_Type *base, *pdm_edma_handle_t* *handle, *status_t* status, void *userData)

PDM eDMA transfer callback function for finish and error.

MCUX_SDK_PDM_EDMA_PDM_ENABLE_INTERNAL

the PDM enable position When calling PDM_TransferReceiveEDMA

struct __pdm_edma_transfer

#include <fsl_pdm_edma.h> PDM edma transfer.

Public Members

volatile uint8_t *data

Data start address to transfer.

volatile size_t dataSize

Total Transfer bytes size.

struct *_pdm_edma_transfer* *linkTransfer

linked transfer configurations

struct *_pdm_edma_handle*

#include <fsl_pdm_edma.h> PDM DMA transfer handle, users should not touch the content of the handle.

Public Members

edma_handle_t *dmaHandle

DMA handler for PDM send

uint8_t count

The transfer data count in a DMA request

uint32_t receivedBytes

total transfer count

uint32_t state

Internal state for PDM eDMA transfer

pdm_edma_callback_t callback

Callback for users while transfer finish or error occurs

bool isLoopTransfer

loop transfer

void *userData

User callback parameter

edma_tcd_t *tcd

TCD pool for eDMA transfer.

uint32_t tcdNum

TCD number

uint32_t tcdUser

Index for user to queue transfer.

uint32_t tcdDriver

Index for driver to get the transfer data and size

volatile uint32_t tcdUsedNum

Index for user to queue transfer.

pdm_edma_multi_channel_interleave_t interleaveType

multi channel transfer interleave type

uint8_t endChannel

The last enabled channel

uint8_t channelNums

total channel numbers

2.59 POWERQUAD: PowerQuad hardware accelerator

void PQ_GetDefaultConfig(*pq_config_t* *config)

Get default configuration.

This function initializes the POWERQUAD configuration structure to a default value. FORMAT register field definitions Bits[15:8] scaler (for scaled 'q31' formats) Bits[5:4] external format. 00b=q15, 01b=q31, 10b=float Bits[1:0] internal format. 00b=q15, 01b=q31, 10b=float POWERQUAD->INAFORMAT = (config->inputAPrescale « 8U) | (config->inputAFormat « 4U) | config->machineFormat

For all Powerquad operations internal format must be float (with the only exception being the FFT related functions, ie FFT/IFFT/DCT/IDCT which must be set to q31). The default values are: config->inputAFormat = kPQ_Float; config->inputAPrescale = 0; config->inputBFormat = kPQ_Float; config->inputBPrescale = 0; config->outputFormat = kPQ_Float; config->outputPrescale = 0; config->tmpFormat = kPQ_Float; config->tmpPrescale = 0; config->machineFormat = kPQ_Float; config->tmpBase = 0xE0000000;

Parameters

- config – Pointer to “pq_config_t” structure.

void PQ_SetConfig(POWERQUAD_Type *base, const *pq_config_t* *config)

Set configuration with format/prescale.

Parameters

- base – POWERQUAD peripheral base address
- config – Pointer to “pq_config_t” structure.

static inline void PQ_SetCoproprocessorScaler(POWERQUAD_Type *base, const *pq_prescale_t* *prescale)

set coprocessor scaler for coprocessor instructions, this function is used to set output saturation and scaling for input/output.

Parameters

- base – POWERQUAD peripheral base address
- prescale – Pointer to “pq_prescale_t” structure.

void PQ_Init(POWERQUAD_Type *base)

Initializes the POWERQUAD module.

Parameters

- base – POWERQUAD peripheral base address.

void PQ_Deinit(POWERQUAD_Type *base)

De-initializes the POWERQUAD module.

Parameters

- base – POWERQUAD peripheral base address.

void PQ_SetFormat(POWERQUAD_Type *base, *pq_computationengine_t* engine, *pq_format_t* format)

Set format for non-coprocessor instructions.

Parameters

- base – POWERQUAD peripheral base address
- engine – Computation engine
- format – Data format

static inline void PQ_WaitDone(POWERQUAD_Type *base)

Wait for the completion.

Parameters

- base – POWERQUAD peripheral base address

static inline void PQ_LnF32(float *pSrc, float *pDst)

Processing function for the floating-point natural log.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_InvF32(float *pSrc, float *pDst)

Processing function for the floating-point reciprocal.

Parameters

- *pSrc – points to the block of input data. The range of the input value is non-zero.
- *pDst – points to the block of output data

static inline void PQ_SqrtF32(float *pSrc, float *pDst)

Processing function for the floating-point square-root.

Parameters

- *pSrc – points to the block of input data. The range of the input value is [0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_InvSqrtF32(float *pSrc, float *pDst)

Processing function for the floating-point inverse square-root.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_EtoxF32(float *pSrc, float *pDst)

Processing function for the floating-point natural exponent.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_EtonxF32(float *pSrc, float *pDst)

Processing function for the floating-point natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_SinF32(float *pSrc, float *pDst)

Processing function for the floating-point sine.

Parameters

- *pSrc – points to the block of input data. The input value is in radians, the range is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_CosF32(float *pSrc, float *pDst)

Processing function for the floating-point cosine.

Parameters

- *pSrc – points to the block of input data. The input value is in radians, the range is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_BiquadF32(float *pSrc, float *pDst)

Processing function for the floating-point biquad.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data

static inline void PQ_DivF32(float *x1, float *x2, float *pDst)

Processing function for the floating-point division.

Get x1 / x2.

Parameters

- x1 – x1
- x2 – x2
- *pDst – points to the block of output data

static inline void PQ_Biquad1F32(float *pSrc, float *pDst)

Processing function for the floating-point biquad.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data

static inline int32_t PQ_LnFixed(int32_t val)

Processing function for the fixed natural log.

Parameters

- val – value to be calculated. The range of the input value is (0 +INFINITY).

Returns

returns ln(val).

static inline int32_t PQ_InvFixed(int32_t val)

Processing function for the fixed reciprocal.

Parameters

- val – value to be calculated. The range of the input value is non-zero.

Returns

returns inv(val).

static inline uint32_t PQ_SqrtFixed(uint32_t val)

Processing function for the fixed square-root.

Parameters

- val – value to be calculated. The range of the input value is [0 +INFINITY).

Returns

returns sqrt(val).

static inline int32_t PQ_InvSqrtFixed(int32_t val)

Processing function for the fixed inverse square-root.

Parameters

- val – value to be calculated. The range of the input value is (0 +INFINITY).

Returns

returns 1/sqrt(val).

static inline int32_t PQ_EtoxFixed(int32_t val)

Processing function for the Fixed natural exponent.

Parameters

- val – value to be calculated. The range of the input value is (-INFINITY +INFINITY).

Returns

returns etox^(val).

static inline int32_t PQ_EtonxFixed(int32_t val)

Processing function for the fixed natural exponent with negative parameter.

Parameters

- val – value to be calculated. The range of the input value is (-INFINITY +INFINITY).

Returns

returns etonx^(val).

static inline int32_t PQ_SinQ31(int32_t val)

Processing function for the fixed sine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q31 format, which means [-pi, pi].

Returns

returns sin(val).

static inline int16_t PQ_SinQ15(int16_t val)

Processing function for the fixed sine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q15 format, which means [-pi, pi].

Returns

returns sin(val).

static inline int32_t PQ_CosQ31(int32_t val)

Processing function for the fixed cosine.

Parameters

- `val` – value to be calculated. The input value is [-1, 1] in Q31 format, which means [-pi, pi].

Returns

returns `cos(val)`.

static inline int16_t PQ_CosQ15(int16_t val)

Processing function for the fixed sine.

Parameters

- `val` – value to be calculated. The input value is [-1, 1] in Q15 format, which means [-pi, pi].

Returns

returns `sin(val)`.

static inline int32_t PQ_BiquadFixed(int32_t val)

Processing function for the fixed biquad.

Parameters

- `val` – value to be calculated

Returns

returns `biquad(val)`.

void PQ_VectorLnF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural log.

Parameters

- `*pSrc` – points to the block of input data
- `*pDst` – points to the block of output data
- `length` – the block of input data.

void PQ_VectorInvF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised reciprocal.

Parameters

- `*pSrc` – points to the block of input data
- `*pDst` – points to the block of output data
- `length` – the block of input data.

void PQ_VectorSqrtF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised square-root.

Parameters

- `*pSrc` – points to the block of input data
- `*pDst` – points to the block of output data
- `length` – the block of input data.

void PQ_VectorInvSqrtF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised inverse square-root.

Parameters

- `*pSrc` – points to the block of input data
- `*pDst` – points to the block of output data
- `length` – the block of input data.

void PQ_VectorEtoxF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorLnFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised natural log.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised reciprocal.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSqrtFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvSqrtFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised inverse square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtoxFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinQ15(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the Q15 vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosQ15(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the Q15 vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinQ31(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosQ31(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorLnFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural log.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised reciprocal.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSqrtFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvSqrtFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised inverse square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtoxFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorBiquadDf2F32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data.

void PQ_VectorBiquadDf2Fixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

void PQ_VectorBiquadDf2Fixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

void PQ_VectorBiquadCascadeDf2F32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

void PQ_VectorBiquadCascadeDf2Fixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

void PQ_VectorBiquadCascadeDf2Fixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

```
int32_t PQ_ArctanFixed(POWERQUAD_Type *base, int32_t x, int32_t y, pq_cordic_iter_t
                      iteration)
```

Processing function for the fixed inverse trigonometric.

Get the inverse tangent, the behavior is like c function atan.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the arctan(0.5), the result of PQ_ArctanFixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24) is more accurate than PQ_ArctanFixed(POWERQUAD, 1, 2, kPQ_Iteration_24).

Parameters

- base – POWERQUAD peripheral base address
- x – value of opposite
- y – value of adjacent
- iteration – iteration times

Returns

The return value is in the range of -2^{26} to 2^{26} , which means $-\pi/2$ to $\pi/2$.

```
int32_t PQ_ArctanhFixed(POWERQUAD_Type *base, int32_t x, int32_t y, pq_cordic_iter_t
                       iteration)
```

Processing function for the fixed inverse trigonometric.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the arctanh(0.5), the result of PQ_ArctanhFixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24) is more accurate than PQ_ArctanhFixed(POWERQUAD, 1, 2, kPQ_Iteration_24).

Parameters

- base – POWERQUAD peripheral base address
- x – value of opposite
- y – value of adjacent
- iteration – iteration times

Returns

The return value is radians, 2^{27} means π . The range is -1.118 to 1.118 radians.

```
int32_t PQ_Arctan2Fixed(POWERQUAD_Type *base, int32_t x, int32_t y, pq_cordic_iter_t
                       iteration)
```

Processing function for the fixed inverse trigonometric.

Get the inverse tangent, it calculates the angle in radians for the quadrant. The behavior is like c function atan2.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the `arctan(0.5)`, the result of `PQ_Arctan2Fixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24)` is more accurate than `PQ_Arctan2Fixed(POWERQUAD, 1, 2, kPQ_Iteration_24)`.

Parameters

- `base` – POWERQUAD peripheral base address
- `x` – value of opposite
- `y` – value of adjacent
- `iteration` – iteration times

Returns

The return value is in the range of -2^{27} to 2^{27} , which means $-\pi$ to π .

`static inline int32_t PQ_Biquad1Fixed(int32_t val)`

Processing function for the fixed biquad.

Parameters

- `val` – value to be calculated

Returns

returns `biquad(val)`.

`void PQ_TransformCFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)`

Processing function for the complex FFT.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – number of input samples
- `pData` – input data
- `pResult` – output data.

`void PQ_TransformRFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)`

Processing function for the real FFT.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – number of input samples
- `pData` – input data
- `pResult` – output data.

`void PQ_TransformIFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)`

Processing function for the inverse complex FFT.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – number of input samples
- `pData` – input data
- `pResult` – output data.

```
void PQ_TransformCDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)
```

Processing function for the complex DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

```
void PQ_TransformRDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)
```

Processing function for the real DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

```
void PQ_TransformIDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)
```

Processing function for the inverse complex DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

```
void PQ_BiquadBackUpInternalState(POWERQUAD_Type *base, int32_t biquad_num, pq_biquad_state_t *state)
```

Processing function for backup biquad context.

Parameters

- base – POWERQUAD peripheral base address
- biquad_num – biquad side
- state – point to states.

```
void PQ_BiquadRestoreInternalState(POWERQUAD_Type *base, int32_t biquad_num, pq_biquad_state_t *state)
```

Processing function for restore biquad context.

Parameters

- base – POWERQUAD peripheral base address
- biquad_num – biquad side
- state – point to states.

```
void PQ_BiquadCascadeDf2Init(pq_biquad_cascade_df2_instance *S, uint8_t numStages, pq_biquad_state_t *pState)
```

Initialization function for the direct form II Biquad cascade filter.

Parameters

- *S – **[inout]** points to an instance of the filter data structure.
- numStages – **[in]** number of 2nd order stages in the filter.
- *pState – **[in]** points to the state buffer.

```
void PQ_BiquadCascadeDf2F32(const pq_biquad_cascade_df2_instance *S, float *pSrc, float  
                           *pDst, uint32_t blockSize)
```

Processing function for the floating-point direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.
- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

```
void PQ_BiquadCascadeDf2Fixed32(const pq_biquad_cascade_df2_instance *S, int32_t *pSrc,  
                               int32_t *pDst, uint32_t blockSize)
```

Processing function for the Q31 direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.
- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

```
void PQ_BiquadCascadeDf2Fixed16(const pq_biquad_cascade_df2_instance *S, int16_t *pSrc,  
                               int16_t *pDst, uint32_t blockSize)
```

Processing function for the Q15 direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.
- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

```
void PQ_FIR(POWERQUAD_Type *base, const void *pAData, int32_t ALength, const void  
            *pBData, int32_t BLength, void *pResult, uint32_t opType)
```

Processing function for the FIR.

Parameters

- base – POWERQUAD peripheral base address
- pAData – the first input sequence
- ALength – number of the first input sequence
- pBData – the second input sequence
- BLength – number of the second input sequence
- pResult – array for the output data
- opType – operation type, could be PQ_FIR_FIR, PQ_FIR_CONVOLUTION, PQ_FIR_CORRELATION.

```
void PQ_FIRIncrement(POWERQUAD_Type *base, int32_t ALength, int32_t BLength, int32_t xOffset)
```

Processing function for the incremental FIR. This function can be used after `pq_fir()` for incremental FIR operation when new x data are available.

Parameters

- `base` – POWERQUAD peripheral base address
- `ALength` – number of input samples
- `BLength` – number of taps
- `xOffset` – offset for number of input samples

```
void PQ_MatrixAddition(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix addition.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – rows and cols for matrix. LENGTH register configuration: `LENGTH[23:16] = M2 cols` `LENGTH[15:8] = M1 cols` `LENGTH[7:0] = M1 rows` This could be constructed using macro `POWERQUAD_MAKE_MATRIX_LEN`.
- `pAData` – input matrix A
- `pBData` – input matrix B
- `pResult` – array for the output data.

```
void PQ_MatrixSubtraction(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix subtraction.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – rows and cols for matrix. LENGTH register configuration: `LENGTH[23:16] = M2 cols` `LENGTH[15:8] = M1 cols` `LENGTH[7:0] = M1 rows` This could be constructed using macro `POWERQUAD_MAKE_MATRIX_LEN`.
- `pAData` – input matrix A
- `pBData` – input matrix B
- `pResult` – array for the output data.

```
void PQ_MatrixMultiplication(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix multiplication.

Parameters

- `base` – POWERQUAD peripheral base address
- `length` – rows and cols for matrix. LENGTH register configuration: `LENGTH[23:16] = M2 cols` `LENGTH[15:8] = M1 cols` `LENGTH[7:0] = M1 rows` This could be constructed using macro `POWERQUAD_MAKE_MATRIX_LEN`.
- `pAData` – input matrix A
- `pBData` – input matrix B

- pResult – array for the output data.

```
void PQ_MatrixProduct(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData,  
void *pResult)
```

Processing function for the matrix product.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pAData – input matrix A
- pBData – input matrix B
- pResult – array for the output data.

```
void PQ_VectorDotProduct(POWERQUAD_Type *base, uint32_t length, void *pAData, void  
*pBData, void *pResult)
```

Processing function for the vector dot product.

Parameters

- base – POWERQUAD peripheral base address
- length – length of vector
- pAData – input vector A
- pBData – input vector B
- pResult – array for the output data.

```
void PQ_MatrixInversion(POWERQUAD_Type *base, uint32_t length, void *pData, void  
*pTmpData, void *pResult)
```

Processing function for the matrix inverse.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pData – input matrix
- pTmpData – input temporary matrix, pTmpData length not less than pData length and 1024 words is sufficient for the largest supported matrix.
- pResult – array for the output data, round down for fixed point.

```
void PQ_MatrixTranspose(POWERQUAD_Type *base, uint32_t length, void *pData, void  
*pResult)
```

Processing function for the matrix transpose.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.

- pData – input matrix
- pResult – array for the output data.

void PQ_MatrixScale(POWERQUAD_Type *base, uint32_t length, float misc, const void *pData, void *pResult)

Processing function for the matrix scale.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- misc – scaling parameters
- pData – input matrix
- pResult – array for the output data.

FSL_POWERQUAD_DRIVER_VERSION

Version.

enum pq_computationengine_t

powerquad computation engine

Values:

enumerator kPQ_CP_PQ

Math engine.

enumerator kPQ_CP_MTX

Matrix engine.

enumerator kPQ_CP_FFT

FFT engine.

enumerator kPQ_CP_FIR

FIR engine.

enumerator kPQ_CP_CORDIC

CORDIC engine.

enum pq_format_t

powerquad data structure format type

Values:

enumerator kPQ_16Bit

Int16 Fixed point.

enumerator kPQ_32Bit

Int32 Fixed point.

enumerator kPQ_Float

Float point.

enum pq_cordic_iter_t

CORDIC iteration.

Values:

enumerator kPQ_Iteration_8

Iterate 8 times.

enumerator kPQ_Iteration_16

Iterate 16 times.

enumerator kPQ_Iteration_24

Iterate 24 times.

typedef struct *_pq_biquad_param* pq_biquad_param_t

Struct to save biquad parameters.

typedef struct *_pq_biquad_state* pq_biquad_state_t

Struct to save biquad state.

typedef union *_pq_float* pq_float_t

Conversion between integer and float type.

PQ_VectorBiquadDf2F32

PQ_VectorBiquadDf2Fixed32

PQ_VectorBiquadDf2Fixed16

PQ_VectorBiquadCascadeDf2F32

PQ_VectorBiquadCascadeDf2Fixed32

PQ_VectorBiquadCascadeDf2Fixed16

PQ_Vector8BiquadDf2CascadeF32

PQ_Vector8BiquadDf2CascadeFixed32

PQ_Vector8BiquadDf2CascadeFixed16

PQ_FLOAT32

PQ_FIXEDPT

CP_PQ

CP_MTX

CP_FFT

CP_FIR

CP_CORDIC

PQ_TRANS

PQ_TRIG

PQ_BIQUAD

PQ_TRANS_FIXED

PQ_TRIG_FIXED

PQ_BIQUAD_FIXED

PQ_INV

PQ_LN

PQ_SQRT
PQ_INVSQRT
PQ_ETOX
PQ_ETONX
PQ_DIV
PQ_SIN
PQ_COS
PQ_BIQ0_CALC
PQ_BIQ1_CALC
PQ_COMP0_ONLY
PQ_COMP1_ONLY
CORDIC_ITER(x)
CORDIC_MIU(x)
CORDIC_T(x)
CORDIC_ARCTAN
CORDIC_ARCTANH
INST_BUSY
PQ_ERRSTAT_OVERFLOW
PQ_ERRSTAT_NAN
PQ_ERRSTAT_FIXEDOVERFLOW
PQ_ERRSTAT_UNDERFLOW
PQ_TRANS_CFFT
PQ_TRANS_IFFT
PQ_TRANS_CDCT
PQ_TRANS_IDCT
PQ_TRANS_RFFT
PQ_TRANS_RDCT
PQ_MTX_SCALE
PQ_MTX_MULT
PQ_MTX_ADD
PQ_MTX_INV
PQ_MTX_PROD
PQ_MTX_SUB

PQ_VEC_DOTP

PQ_MTX_TRAN

PQ_FIR_FIR

PQ_FIR_CONVOLUTION

PQ_FIR_CORRELATION

PQ_FIR_INCREMENTAL

_pq_ln0(x)

_pq_inv0(x)

_pq_sqrt0(x)

_pq_invsqrt0(x)

_pq_etox0(x)

_pq_etonx0(x)

_pq_sin0(x)

_pq_cos0(x)

_pq_biquad0(x)

_pq_ln_fx0(x)

_pq_inv_fx0(x)

_pq_sqrt_fx0(x)

_pq_invsqrt_fx0(x)

_pq_etox_fx0(x)

_pq_etonx_fx0(x)

_pq_sin_fx0(x)

_pq_cos_fx0(x)

_pq_biquad0_fx(x)

_pq_div0(x)

_pq_div1(x)

_pq_ln1(x)

_pq_inv1(x)

_pq_sqrt1(x)

_pq_invsqrt1(x)

_pq_etox1(x)

_pq_etonx1(x)

_pq_sin1(x)

`_pq_cos1(x)`

`_pq_biquad1(x)`

`_pq_ln_fx1(x)`

`_pq_inv_fx1(x)`

`_pq_sqrt_fx1(x)`

`_pq_invsqrt_fx1(x)`

`_pq_etox_fx1(x)`

`_pq_etonx_fx1(x)`

`_pq_sin_fx1(x)`

`_pq_cos_fx1(x)`

`_pq_biquad1_fx(x)`

`_pq_readMult0()`

`_pq_readAdd0()`

`_pq_readMult1()`

`_pq_readAdd1()`

`_pq_readMult0_fx()`

`_pq_readAdd0_fx()`

`_pq_readMult1_fx()`

`_pq_readAdd1_fx()`

`PQ_LN_INF`

Parameter used for vector $\ln(x)$

`PQ_INV_INF`

Parameter used for vector $1/x$

`PQ_SQRT_INF`

Parameter used for vector $\sqrt{x}$

`PQ_ISQRT_INF`

Parameter used for vector $1/\sqrt{x}$

`PQ_ETOX_INF`

Parameter used for vector e^x

`PQ_ETONX_INF`

Parameter used for vector e^{-x}

`PQ_SIN_INF`

Parameter used for vector $\sin(x)$

`PQ_COS_INF`

Parameter used for vector $\cos(x)$

`PQ_RUN_OPCODE_R3_R2(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R5_R4(BATCH_OPCODE, BATCH_MACHINE)`

PQ_RUN_OPCODE_R7_R6(BATCH_OPCODE, BATCH_MACHINE)

PQ_Vector8_FP(middle, last, BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

PQ_RUN_OPCODE_R2_R3(BATCH_OPCODE, BATCH_MACHINE)

PQ_RUN_OPCODE_R4_R5(BATCH_OPCODE, BATCH_MACHINE)

PQ_RUN_OPCODE_R6_R7(BATCH_OPCODE, BATCH_MACHINE)

PQ_Vector8_FX(middle, last, BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

PQ_Initiate_Vector_Func(pSrc, pDst)

Start 32-bit data vector calculation.

Start the vector calculation, the input data could be float, int32_t or Q31.

Parameters

- pSrc – Pointer to the source data.
- pDst – Pointer to the destination data.

PQ_End_Vector_Func()

End vector calculation.

This function should be called after vector calculation.

PQ_StartVector(PSRC, PDST, LENGTH)

Start 32-bit data vector calculation.

Start the vector calculation, the input data could be float, int32_t or Q31.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_StartVectorFixed16(PSRC, PDST, LENGTH)

Start 16-bit data vector calculation.

Start the vector calculation, the input data could be int16_t. This function should be use with PQ_Vector8Fixed16.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_StartVectorQ15(PSRC, PDST, LENGTH)

Start Q15-bit data vector calculation.

Start the vector calculation, the input data could be Q15. This function should be use with PQ_Vector8Q15. This function is dedicate for SinQ15/CosQ15 vector calculation. Because PowerQuad only supports Q31 Sin/Cos fixed function, so the input Q15 data is left shift 16 bits first, after Q31 calculation, the output data is right shift 16 bits.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_EndVector()

End vector calculation.

This function should be called after vector calculation.

PQ_Vector8F32(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Float data vector calculation.

Float data vector calculation, the input data should be float. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
float output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Fixed32(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Fixed 32bits data vector calculation.

Float data vector calculation, the input data should be 32-bit integer. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF, PQ_SIN_INF, PQ_COS_INF. When this function is used for sin/cos calculation, the input data should be in the format Q1.31. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 4, 9, 16, 25, 36, 49, 64};
int32_t output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Fixed16(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Fixed 32bits data vector calculation.

Float data vector calculation, the input data should be 16-bit integer. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {1, 4, 9, 16, 25, 36, 49, 64};
int16_t output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Q15(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Q15 data vector calculation.

Q15 data vector calculation, this function should only be used for sin/cos Q15 calculation, and the coprocessor output prescaler must be set to 31 before this function. This function loads Q15 data and left shift 16 bits, calculate and right shift 16 bits, then stores to the output array. The input range -1 to 1 means -pi to pi. For example, to calculate sin of a vector, use like this:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {...}
```

(continues on next page)

(continued from previous page)

```

int16_t output[VECTOR_LEN];
const pq_prescale_t prescale =
{
    .inputPrescale = 0,
    .outputPrescale = 31,
    .outputSaturate = 0
};

PQ_SetCoproprocessorScaler(POWERQUAD, const pq_prescale_t *prescale);

PQ_StartVectorQ15(pSrc, pDst, length);
PQ_Vector8Q15(PQ_SQRT_INF);
PQ_EndVector();

```

PQ_DF2_Vector8_FP(middle, last)

Float data vector biquad direct form II calculation.

Biquad filter, the input and output data are float data. Biquad side 0 is used. Example:

```

#define VECTOR_LEN 16
float input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_Initiate_Vector_Func(pSrc,pDst);
PQ_DF2_Vector8_FP(false,false);
PQ_DF2_Vector8_FP(true,true);
PQ_End_Vector_Func();

```

PQ_DF2_Vector8_FX(middle, last)

Fixed data vector biquad direct form II calculation.

Biquad filter, the input and output data are fixed data. Biquad side 0 is used. Example:

```

#define VECTOR_LEN 16
int32_t input[VECTOR_LEN] = {1024, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

```

(continues on next page)

(continued from previous page)

```
PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);
```

```
PQ_Initiate_Vector_Func(pSrc,pDst);
```

```
PQ_DF2_Vector8_FX(false,false);
```

```
PQ_DF2_Vector8_FX(true,true);
```

```
PQ_End_Vector_Func();
```

PQ_Vector8BiquadDf2F32()

Float data vector biquad direct form II calculation.

Biquad filter, the input and output data are float data. Biquad side 0 is used. Example:

```
#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
float output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2F32();
PQ_EndVector();
```

PQ_Vector8BiquadDf2Fixed32()

Fixed 32-bit data vector biquad direct form II calculation.

Biquad filter, the input and output data are Q31 or 32-bit integer. Biquad side 0 is used. Example:

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2Fixed32();
PQ_EndVector();
```

PQ_Vector8BiquadDf2Fixed16()

Fixed 16-bit data vector biquad direct form II calculation.

Biquad filter, the input and output data are Q15 or 16-bit integer. Biquad side 0 is used.
Example:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int16_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2Fixed16();
PQ_EndVector();
```

PQ_DF2_Cascade_Vector8_FP(middle, last)

Float data vector direct form II biquad cascade filter.

The input and output data are float data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 16
float input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_Initiate_Vector_Func(pSrc, pDst);
PQ_DF2_Cascade_Vector8_FP(false, false);
```

(continues on next page)

(continued from previous page)

```
PQ_DF2_Cascade_Vector8_FP(true, true);
PQ_End_Vector_Func();
```

PQ_DF2_Cascade_Vector8_FX(middle, last)

Fixed data vector direct form II biquad cascade filter.

The input and output data are fixed data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 16
int32_t input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_Initiate_Vector_Func(pSrc, pDst);
PQ_DF2_Cascade_Vector8_FX(false, false);
PQ_DF2_Cascade_Vector8_FX(true, true);
PQ_End_Vector_Func();
```

PQ_Vector8BiquadDf2CascadeF32()

Float data vector direct form II biquad cascade filter.

The input and output data are float data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
float output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
```

(continues on next page)

(continued from previous page)

```

        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2CascadeF32();
PQ_EndVector();

```

PQ_Vector8BiquadDf2CascadeFixed32()

Fixed 32-bit data vector direct form II biquad cascade filter.

The input and output data are fixed 32-bit data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```

#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2CascadeFixed32();
PQ_EndVector();

```

PQ_Vector8BiquadDf2CascadeFixed16()

Fixed 16-bit data vector direct form II biquad cascade filter.

The input and output data are fixed 16-bit data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2CascadeFixed16();
PQ_EndVector();
```

POWERQUAD_MAKE_MATRIX_LEN(mat1Row, mat1Col, mat2Col)

Make the length used for matrix functions.

PQ_Q31_2_FLOAT(x)

Convert Q31 to float.

PQ_Q15_2_FLOAT(x)

Convert Q15 to float.

struct pq_prescale_t

#include <fsl_powerquad.h> Coprocessor prescale.

Public Members

int8_t inputPrescale

Input prescale.

int8_t outputPrescale

Output prescale.

int8_t outputSaturate

Output saturate at n bits, for example 0x11 is 8 bit space, the value will be truncated at +127 or -128.

struct pq_config_t

#include <fsl_powerquad.h> powerquad data structure format

Public Members

pq_format_t inputAFormat

Input A format.

int8_t inputAPrescale

Input A prescale, for example 1.5 can be $1.5 \cdot 2^n$ if you scale by 'shifting' ('scaling' by a factor of n).

pq_format_t inputBFormat

Input B format.

int8_t inputBPrescale

Input B prescale.

pq_format_t outputFormat

Out format.

int8_t outputPrescale

Out prescale.

pq_format_t tmpFormat

Temp format.

int8_t tmpPrescale

Temp prescale.

pq_format_t machineFormat

Machine format.

uint32_t *tmpBase

Tmp base address.

struct __pq_biquad_param

#include <fsl_powerquad.h> Struct to save biquad parameters.

Public Members

float v_n_1

v[n-1], set to 0 when initialization.

float v_n

v[n], set to 0 when initialization.

float a_1

a[1]

float a_2

a[2]

float b_0

b[0]

```
float b_1
    b[1]
float b_2
    b[2]
struct __pq_biquad_state
    #include <fsl_powerquad.h> Struct to save biquad state.
```

Public Members

```
pq_biquad_param_t param
    Filter parameter.
uint32_t compreg
    Internal register, set to 0 when initialization.
struct pq_biquad_cascade_df2_instance
    #include <fsl_powerquad.h> Instance structure for the direct form II Biquad cascade filter.
```

Public Members

```
uint8_t numStages
    Number of 2nd order stages in the filter.
pq_biquad_state_t *pState
    Points to the array of state coefficients.
union __pq_float
    #include <fsl_powerquad.h> Conversion between integer and float type.
```

Public Members

```
float floatX
    Float type.
uint32_t integerX
    Unsigned interger type.
```

2.60 PXP: Pixel Pipeline

```
void PXP_Init(PXP_Type *base)
    Initialize the PXP.
    This function enables the PXP peripheral clock, and resets the PXP registers to default status.
```

Parameters

- base – PXP peripheral base address.

```
void PXP_Deinit(PXP_Type *base)
    De-initialize the PXP.
    This function disables the PXP peripheral clock.
```

Parameters

- base – PXP peripheral base address.

`void PXP__Reset(PXP_Type *base)`

Reset the PXP.

This function resets the PXP peripheral registers to default status.

Parameters

- `base` – PXP peripheral base address.

`void PXP__ResetControl(PXP_Type *base)`

Reset the PXP and the control register to initialized state.

Parameters

- `base` – PXP peripheral base address.

`static inline void PXP__Start(PXP_Type *base)`

Start process.

Start PXP process using current configuration.

Parameters

- `base` – PXP peripheral base address.

`static inline void PXP__EnableLcdHandShake(PXP_Type *base, bool enable)`

Enable or disable LCD hand shake.

Parameters

- `base` – PXP peripheral base address.
- `enable` – True to enable, false to disable.

`static inline void PXP__EnableContinuousRun(PXP_Type *base, bool enable)`

Enable or disable continuous run.

If continuous run not enabled, `PXP__Start` starts the PXP process. When completed, PXP enters idle mode and flag `kPXP_CompleteFlag` asserts.

If continuous run enabled, the PXP will repeat based on the current configuration register settings.

Parameters

- `base` – PXP peripheral base address.
- `enable` – True to enable, false to disable.

`static inline void PXP__SetProcessBlockSize(PXP_Type *base, pxp_block_size_t size)`

Set the PXP processing block size.

This function chooses the pixel block size that PXP using during process. Larger block size means better performance, but be careful that when PXP is rotating, the output must be divisible by the block size selected.

Parameters

- `base` – PXP peripheral base address.
- `size` – The pixel block size.

`static inline void PXP__EnableProcessEngine(PXP_Type *base, uint32_t mask, bool enable)`

Enables or disables PXP engines in the process flow.

Parameters

- `base` – PXP peripheral base address.
- `mask` – The engines to enable. Logical OR of `pxp_process_engine_name_t`.

- enable – true to enable, false to disable.

static inline uint32_t PXP_GetStatusFlags(PXP_Type *base)

Gets PXP status flags.

This function gets all PXP status flags. The flags are returned as the logical OR value of the enumerators `_pxp_flags`. To check a specific status, compare the return value with enumerators in `_pxp_flags`. For example, to check whether the PXP has completed process, use like this:

```
if (kPXP_CompleteFlag & PXP_GetStatusFlags(PXP))
{
    ...
}
```

Parameters

- base – PXP peripheral base address.

Returns

PXP status flags which are OR'ed by the enumerators in the `_pxp_flags`.

static inline void PXP_ClearStatusFlags(PXP_Type *base, uint32_t statusMask)

Clears status flags with the provided mask.

This function clears PXP status flags with a provided mask.

Parameters

- base – PXP peripheral base address.
- statusMask – The status flags to be cleared; it is logical OR value of `_pxp_flags`.

static inline uint8_t PXP_GetAxiErrorId(PXP_Type *base, uint8_t axiIndex)

Gets the AXI ID of the failing bus operation.

Parameters

- base – PXP peripheral base address.
- axiIndex – Which AXI to get
 - 0: AXI0
 - 1: AXI1

Returns

The AXI ID of the failing bus operation.

static inline void PXP_EnableInterrupts(PXP_Type *base, uint32_t mask)

Enables PXP interrupts according to the provided mask.

This function enables the PXP interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_pxp_interrupt_enable`. For example, to enable PXP process complete interrupt and command loaded interrupt, do the following.

```
PXP_EnableInterrupts(PXP, kPXP_CommandLoadInterruptEnable | kPXP_
    ↪ CompleteInterruptEnable);
```

Parameters

- base – PXP peripheral base address.
- mask – The interrupts to enable. Logical OR of `_pxp_interrupt_enable`.

```
static inline void PXP_DisableInterrupts(PXP_Type *base, uint32_t mask)
```

Disables PXP interrupts according to the provided mask.

This function disables the PXP interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_pxp_interrupt_enable`.

Parameters

- `base` – PXP peripheral base address.
- `mask` – The interrupts to disable. Logical OR of `_pxp_interrupt_enable`.

```
void PXP_SetAlphaSurfaceBufferConfig(PXP_Type *base, const pxp_as_buffer_config_t *config)
```

Set the alpha surface input buffer configuration.

Parameters

- `base` – PXP peripheral base address.
- `config` – Pointer to the configuration.

```
void PXP_SetAlphaSurfaceBlendConfig(PXP_Type *base, const pxp_as_blend_config_t *config)
```

Set the alpha surface blending configuration.

Parameters

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

```
void PXP_SetAlphaSurfaceBlendSecondaryConfig(PXP_Type *base, const  
                                              pxp_as_blend_secondary_config_t *config)
```

Set the alpha surface blending configuration for the secondary engine.

Parameters

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

```
void PXP_SetAlphaSurfaceOverlayColorKey(PXP_Type *base, uint8_t num, uint32_t colorKeyLow,  
                                         uint32_t colorKeyHigh)
```

Set the alpha surface overlay color key.

If a pixel in the current overlay image with a color that falls in the range from the `colorKeyLow` to `colorKeyHigh` range, it will use the process surface pixel value for that location. If no PS image is present or if the PS image also matches its colorkey range, the PS background color is used.

Note: Colorkey operations are higher priority than alpha or ROP operations

Parameters

- `base` – PXP peripheral base address.
- `num` – instance number. 0 for alpha engine A, 1 for alpha engine B.
- `colorKeyLow` – Color key low range.
- `colorKeyHigh` – Color key high range.

```
static inline void PXP_EnableAlphaSurfaceOverlayColorKey(PXP_Type *base, uint32_t num, bool  
                                                         enable)
```

Enable or disable the alpha surface color key.

Parameters

- base – PXP peripheral base address.
- num – instance number. 0 for alpha engine A, 1 for alpha engine B.
- enable – True to enable, false to disable.

```
void PXP_SetAlphaSurfacePosition(PXP_Type *base, uint16_t upperLeftX, uint16_t upperLeftY,  
                                uint16_t lowerRightX, uint16_t lowerRightY)
```

Set the alpha surface position in output buffer.

Parameters

- base – PXP peripheral base address.
- upperLeftX – X of the upper left corner.
- upperLeftY – Y of the upper left corner.
- lowerRightX – X of the lower right corner.
- lowerRightY – Y of the lower right corner.

```
static inline void PXP_SetProcessSurfaceBackGroundColor(PXP_Type *base, uint8_t num,  
                                                         uint32_t backGroundColor)
```

Set the back ground color of PS.

Parameters

- base – PXP peripheral base address.
- num – instance number. 0 for alpha engine A, 1 for alpha engine B.
- backGroundColor – Pixel value of the background color.

```
void PXP_SetProcessSurfaceBufferConfig(PXP_Type *base, const pxp_ps_buffer_config_t *config)
```

Set the process surface input buffer configuration.

Parameters

- base – PXP peripheral base address.
- config – Pointer to the configuration.

```
void PXP_SetProcessSurfaceScaler(PXP_Type *base, uint16_t inputWidth, uint16_t inputHeight,  
                                uint16_t outputWidth, uint16_t outputHeight)
```

Set the process surface scaler configuration.

The valid down scale fact is $1/(2^{12}) \sim 16$.

Parameters

- base – PXP peripheral base address.
- inputWidth – Input image width.
- inputHeight – Input image height.
- outputWidth – Output image width.
- outputHeight – Output image height.

```
void PXP_SetProcessSurfacePosition(PXP_Type *base, uint16_t upperLeftX, uint16_t upperLeftY,  
                                   uint16_t lowerRightX, uint16_t lowerRightY)
```

Set the process surface position in output buffer.

Parameters

- base – PXP peripheral base address.
- upperLeftX – X of the upper left corner.
- upperLeftY – Y of the upper left corner.

- lowerRightX – X of the lower right corner.
- lowerRightY – Y of the lower right corner.

```
void PXP_SetProcessSurfaceColorKey(PXP_Type *base, uint8_t num, uint32_t colorKeyLow,
                                   uint32_t colorKeyHigh)
```

Set the process surface color key.

If the PS image matches colorkey range, the PS background color is output. Set colorKeyLow to 0xFFFFFFFF and p colorKeyHigh to 0 will disable the colorkeying.

Parameters

- base – PXP peripheral base address.
- num – instance number. 0 for alpha engine A, 1 for alpha engine B.
- colorKeyLow – Color key low range.
- colorKeyHigh – Color key high range.

```
static inline void PXP_SetProcessSurfaceYUVFormat(PXP_Type *base, pxp_ps_yuv_format_t
                                                  format)
```

Set the process surface input pixel format YUV or YCbCr.

If process surface input pixel format is YUV and CSC1 is not enabled, in other words, the process surface output pixel format is also YUV, then this function should be called to set whether input pixel format is YUV or YCbCr.

Parameters

- base – PXP peripheral base address.
- format – The YUV format.

```
void PXP_SetOutputBufferConfig(PXP_Type *base, const pxp_output_buffer_config_t *config)
```

Set the PXP output buffer configuration.

Parameters

- base – PXP peripheral base address.
- config – Pointer to the configuration.

```
static inline void PXP_SetOverwrittenAlphaValue(PXP_Type *base, uint8_t alpha)
```

Set the global overwritten alpha value.

If global overwritten alpha is enabled, the alpha component in output buffer pixels will be overwritten, otherwise the computed alpha value is used.

Parameters

- base – PXP peripheral base address.
- alpha – The alpha value.

```
static inline void PXP_EnableOverWrittenAlpha(PXP_Type *base, bool enable)
```

Enable or disable the global overwritten alpha value.

If global overwritten alpha is enabled, the alpha component in output buffer pixels will be overwritten, otherwise the computed alpha value is used.

Parameters

- base – PXP peripheral base address.
- enable – True to enable, false to disable.

```
static inline void PXP_SetRotateConfig(PXP_Type *base, pxp_rotate_position_t position,
                                       pxp_rotate_degree_t degree, pxp_flip_mode_t flipMode)
```

Set the rotation configuration.

The PXP could rotate the process surface or the output buffer. There are two PXP versions:

- Version 1: Only has one rotate sub module, the output buffer and process surface share the same rotate sub module, which means the process surface and output buffer could not be rotate at the same time. When pass in `kPXP_RotateOutputBuffer`, the process surface could not use the rotate, Also when pass in `kPXP_RotateProcessSurface`, output buffer could not use the rotate.
- Version 2: Has two separate rotate sub modules, the output buffer and process surface could configure the rotation independently.

Upper layer could use the macro `PXP_SHARE_ROTATE` to check which version is. `PXP_SHARE_ROTATE=1` means version 1.

Note: This function is different depends on the macro `PXP_SHARE_ROTATE`.

Parameters

- `base` – PXP peripheral base address.
- `position` – Rotate process surface or output buffer.
- `degree` – Rotate degree.
- `flipMode` – Flip mode.

```
void PXP_BuildRect(PXP_Type *base, pxp_output_pixel_format_t outFormat, uint32_t value,
                  uint16_t width, uint16_t height, uint16_t pitch, uint32_t outAddr)
```

Build a solid rectangle of given pixel value.

Parameters

- `base` – PXP peripheral base address.
- `outFormat` – output pixel format.
- `value` – The value of the pixel to be filled in the rectangle in ARGB8888 format.
- `width` – width of the rectangle.
- `height` – height of the rectangle.
- `pitch` – output pitch in byte.
- `outAddr` – address of the memory to store the rectangle.

```
void PXP_SetNextCommand(PXP_Type *base, void *commandAddr)
```

Set the next command.

The PXP supports a primitive ability to queue up one operation while the current operation is running. Workflow:

- Prepare the PXP register values except `STAT`, `CSCCOEFn`, `NEXT` in the memory in the order they appear in the register map.
- Call this function sets the new operation to PXP.
- There are two methods to check whether the PXP has loaded the new operation. The first method is using `PXP_IsNextCommandPending`. If there is new operation not loaded by the PXP, this function returns true. The second method is checking the flag

kPXP_CommandLoadFlag, if command loaded, this flag asserts. User could enable interrupt kPXP_CommandLoadInterruptEnable to get the loaded signal in interrupt way.

d. When command loaded by PXP, a new command could be set using this function.

```
uint32_t pxp_command1[48];
uint32_t pxp_command2[48];

pxp_command1[0] = ...;
pxp_command1[1] = ...;
...
pxp_command2[0] = ...;
pxp_command2[1] = ...;
...

while (PXP_IsNextCommandPending(PXP))
{
}

PXP_SetNextCommand(PXP, pxp_command1);

while (PXP_IsNextCommandPending(PXP))
{
}

PXP_SetNextCommand(PXP, pxp_command2);
```

Parameters

- base – PXP peripheral base address.
- commandAddr – Address of the new command.

static inline bool PXP_IsNextCommandPending(PXP_Type *base)

Check whether the next command is pending.

Parameters

- base – UART peripheral base address.

Returns

True is pending, false is not.

static inline void PXP_CancelNextCommand(PXP_Type *base)

Cancel command set by PXP_SetNextCommand.

Parameters

- base – UART peripheral base address.

void PXP_SetCsc1Mode(PXP_Type *base, *pxp_csc1_mode_t* mode)

Set the CSC1 mode.

The CSC1 module receives scaled YUV/YCbCr444 pixels from the scale engine and converts the pixels to the RGB888 color space. It could only be used by process surface.

Parameters

- base – PXP peripheral base address.
- mode – The conversion mode.

static inline void PXP_EnableCsc1(PXP_Type *base, bool enable)

Enable or disable the CSC1.

Parameters

- base – PXP peripheral base address.

- enable – True to enable, false to disable.

```
void PXP_SetInternalRamData(PXP_Type *base, pxp_ram_t ram, uint16_t bytesNum, uint8_t *data, uint16_t memStartAddr)
```

Write data to the PXP internal memory.

Parameters

- base – PXP peripheral base address.
- ram – Which internal memory to write.
- bytesNum – How many bytes to write.
- data – Pointer to the data to write.
- memStartAddr – The start address in the internal memory to write the data.

```
void PXP_SetDitherFinalLutData(PXP_Type *base, const pxp_dither_final_lut_data_t *data)
```

Set the dither final LUT data.

The dither final LUT is only applicable to dither engine 0. It takes the bits[7:4] of the output pixel and looks up and 8 bit value from the 16 value LUT to generate the final output pixel to the next process module.

Parameters

- base – PXP peripheral base address.
- data – Pointer to the LUT data to set.

```
static inline void PXP_SetDitherConfig(PXP_Type *base, const pxp_dither_config_t *config)
```

Set the configuration for the dither block.

If the pre-dither LUT, post-dither LUT or ordered dither is used, please call PXP_SetInternalRamData to set the LUT data to internal memory.

If the final LUT is used, please call PXP_SetDitherFinalLutData to set the LUT data.

Note: When using ordered dithering, please set the PXP process block size same with the ordered dithering matrix size using function PXP_SetProcessBlockSize.

Parameters

- base – PXP peripheral base address.
- config – Pointer to the configuration.

```
void PXP_EnableDither(PXP_Type *base, bool enable)
```

Enable or disable dither engine in the PXP process path.

After the initialize function PXP_Init, the dither engine is disabled and not use in the PXP processing path. This function enables the dither engine and routes the dither engine output to the output buffer. When the dither engine is enabled using this function, PXP_SetDitherConfig must be called to configure dither engine correctly, otherwise there is not output to the output buffer.

Parameters

- base – PXP peripheral base address.
- enable – Pass in true to enable, false to disable.

```
void PXP_SetPorterDuffConfig(PXP_Type *base, uint8_t num, const pxp_porter_duff_config_t *config)
```

Set the Porter Duff configuration for one of the alpha process engine.

Parameters

- base – PXP peripheral base address.
- num – instance number.
- config – Pointer to the configuration.

status_t PXP_GetPorterDuffConfigExt(*pxp_porter_duff_blend_mode_t* mode,
pxp_porter_duff_config_t *config, uint8_t
dstGlobalAlphaMode, uint8_t dstAlphaMode, uint8_t
dstColorMode, uint8_t srcGlobalAlphaMode, uint8_t
srcAlphaMode, uint8_t srcColorMode, uint8_t
dstGlobalAlpha, uint8_t srcGlobalAlpha)

Get the Porter Duff configuration.

The FactorMode are selected based on blend mode, the other values are set based on input parameters. These values could be modified after calling this function. This function is extened PXP_GetPorterDuffConfig.

Parameters

- mode – The blend mode.
- config – Pointer to the configuration.
- dstGlobalAlphaMode – Destination layer (or PS, s0) global alpha mode, see *pxp_porter_duff_global_alpha_mode*
- dstAlphaMode – Destination layer (or PS, s0) alpha mode, see *pxp_porter_duff_alpha_mode*.
- dstColorMode – Destination layer (or PS, s0) color mode, see *pxp_porter_duff_color_mode*.
- srcGlobalAlphaMode – Source layer (or AS, s1) global alpha mode, see *pxp_porter_duff_global_alpha_mode*
- srcAlphaMode – Source layer (or AS, s1) alpha mode, see *pxp_porter_duff_alpha_mode*.
- srcColorMode – Source layer (or AS, s1) color mode, see *pxp_porter_duff_color_mode*.
- dstGlobalAlpha – Destination layer (or PS, s0) global alpha value, 0~255
- srcGlobalAlpha – Source layer (or AS, s1) global alpha value, 0~255

Return values

- kStatus_Success – Successfully get the configuratoin.
- kStatus_InvalidArgument – The blend mode not supported.

static inline *status_t* PXP_GetPorterDuffConfig(*pxp_porter_duff_blend_mode_t* mode,
pxp_porter_duff_config_t *config)

Get the Porter Duff configuration by blend mode.

The FactorMode are selected based on blend mode, the AlphaMode are set to kPXP_PorterDuffAlphaStraight, the ColorMode are set to kPXP_PorterDuffColorWithAlpha, the GlobalAlphaMode are set to kPXP_PorterDuffLocalAlpha. These values could be modified after calling this function.

Parameters

- mode – The blend mode.
- config – Pointer to the configuration.

Return values

- kStatus_Success – Successfully get the configuratoin.
- kStatus_InvalidArgument – The blend mode not supported.

FSL_PXP_DRIVER_VERSION

enum __pxp_interrupt_enable

PXP interrupts to enable.

Values:

enumerator kPXP_CompleteInterruptEnable

PXP process completed. bit 1

enumerator kPXP_CommandLoadInterruptEnable

Interrupt to show that the command set by PXP_SetNextCommand has been loaded.
bit 2

enumerator kPXP_CompressDoneInterruptEnable

Compress done interrupt enable. bit 15

enumerator kPXP_InputFetchCh0InterruptEnable

Input fetch channel 0 completed. bit 16

enumerator kPXP_InputFetchCh1InterruptEnable

Input fetch channel 1 completed. bit 17

enumerator kPXP_InputStoreCh0InterruptEnable

Input store channel 0 completed. bit 18

enumerator kPXP_InputStoreCh1InterruptEnable

Input store channel 1 completed. bit 19

enumerator kPXP_DitherFetchCh0InterruptEnable

Dither fetch channel 0 completed. bit 20

enumerator kPXP_DitherFetchCh1InterruptEnable

Dither fetch channel 1 completed. bit 21

enumerator kPXP_DitherStoreCh0InterruptEnable

Dither store channle 0 completed. bit 22

enumerator kPXP_DitherStoreCh1InterruptEnable

Dither store channle 1 completed. bit 23

enumerator kPXP_WfeaStoreCh0InterruptEnable

WFE-A store channel 0 completed. bit 24

enumerator kPXP_WfeaStoreCh1InterruptEnable

WFE-A store channel 1 completed. bit 25

enumerator kPXP_WfebStoreCh0InterruptEnable

WFE-B store channel 0 completed. bit 26

enumerator kPXP_WfebStoreCh1InterruptEnable

WFE-B store channel 1 completed. bit 27

enumerator kPXP_InputStoreInterruptEnable

Input store completed. bit 28

enumerator kPXP_DitherStoreInterruptEnable

Dither store completed. bit 29

enumerator kPXP_WfeaStoreInterruptEnable
WFE-A store completed. bit 30

enumerator kPXP_WfebStoreInterruptEnable
WFE-B store completed. bit 31

enum __pxp_flags
PXP status flags.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator kPXP_CompleteFlag
PXP process completed. bit 0

enumerator kPXP_Axi0WriteErrorFlag
PXP encountered an AXI write error and processing has been terminated. bit 1

enumerator kPXP_Axi0ReadErrorFlag
PXP encountered an AXI read error and processing has been terminated. bit 2

enumerator kPXP_CommandLoadFlag
The command set by PXP_SetNextCommand has been loaded, could set new command.
bit 3

enumerator kPXP_CompressDoneFlag
Compress done. bit 15

enumerator kPXP_InputFetchCh0CompleteFlag
Input fetch channel 0 completed. bit 16

enumerator kPXP_InputFetchCh1CompleteFlag
Input fetch channel 1 completed. bit 17

enumerator kPXP_InputStoreCh0CompleteFlag
Input store channel 0 completed. bit 18

enumerator kPXP_InputStoreCh1CompleteFlag
Input store channel 1 completed. bit 19

enumerator kPXP_DitherFetchCh0CompleteFlag
Dither fetch channel 0 completed. bit 20

enumerator kPXP_DitherFetchCh1CompleteFlag
Dither fetch channel 1 completed. bit 21

enumerator kPXP_DitherStoreCh0CompleteFlag
Dither store channel 0 completed. bit 22

enumerator kPXP_DitherStoreCh1CompleteFlag
Dither store channel 1 completed. bit 23

enumerator kPXP_WfeaStoreCh0CompleteFlag
WFE-A store channel 0 completed. bit 24

enumerator kPXP_WfeaStoreCh1CompleteFlag
WFE-A store channel 1 completed. bit 25

enumerator kPXP_WfebStoreCh0CompleteFlag
WFE-B store channel 0 completed. bit 26

enumerator kPXP_WfebStoreCh1CompleteFlag
WFE-B store channel 1 completed. bit 27

enumerator kPXP_InputStoreCompleteFlag
Input store completed. bit 28

enumerator kPXP_DitherStoreCompleteFlag
Dither store completed. bit 29

enumerator kPXP_WfeaStoreCompleteFlag
WFE-A store completed. bit 30

enumerator kPXP_WfebStoreCompleteFlag
WFE-B store completed. bit 31

enum __pxp_flip_mode
PXP output flip mode.

Values:

enumerator kPXP_FlipDisable
Flip disable.

enumerator kPXP_FlipHorizontal
Horizontal flip.

enumerator kPXP_FlipVertical
Vertical flip.

enumerator kPXP_FlipBoth
Flip both directions.

enum __pxp_rotate_position
PXP rotate mode.

Values:

enumerator kPXP_RotateOutputBuffer
Rotate the output buffer.

enumerator kPXP_RotateProcessSurface
Rotate the process surface. Cannot be used together with flip, scale, or decimation function.

enum __pxp_rotate_degree
PXP rotate degree.

Values:

enumerator kPXP_Rotate0
Clock wise rotate 0 deg.

enumerator kPXP_Rotate90
Clock wise rotate 90 deg.

enumerator kPXP_Rotate180
Clock wise rotate 180 deg.

enumerator kPXP_Rotate270
Clock wise rotate 270 deg.

enum __pxp_interlaced_output_mode
PXP interlaced output mode.

Values:

enumerator kPXP_OutputProgressive

All data written in progressive format to output buffer 0.

enumerator kPXP_OutputField0

Only write field 0 data to output buffer 0.

enumerator kPXP_OutputField1

Only write field 1 data to output buffer 0.

enumerator kPXP_OutputInterlaced

Field 0 write to buffer 0, field 1 write to buffer 1.

enum __pxp_output_pixel_format

PXP output buffer format.

Values:

enumerator kPXP_OutputPixelFormatARGB8888

32-bit pixels with alpha.

enumerator kPXP_OutputPixelFormatRGB888

32-bit pixels without alpha (unpacked 24-bit format)

enumerator kPXP_OutputPixelFormatRGB888P

24-bit pixels without alpha (packed 24-bit format)

enumerator kPXP_OutputPixelFormatARGB1555

16-bit pixels with alpha.

enumerator kPXP_OutputPixelFormatARGB4444

16-bit pixels with alpha.

enumerator kPXP_OutputPixelFormatRGB555

16-bit pixels without alpha.

enumerator kPXP_OutputPixelFormatRGB444

16-bit pixels without alpha.

enumerator kPXP_OutputPixelFormatRGB565

16-bit pixels without alpha.

enumerator kPXP_OutputPixelFormatYUV1P444

32-bit pixels (1-plane XYUV unpacked).

enumerator kPXP_OutputPixelFormatUYVY1P422

16-bit pixels (1-plane U0,Y0,V0,Y1 interleaved bytes)

enumerator kPXP_OutputPixelFormatVYUY1P422

16-bit pixels (1-plane V0,Y0,U0,Y1 interleaved bytes)

enumerator kPXP_OutputPixelFormatY8

8-bit monochrome pixels (1-plane Y luma output)

enumerator kPXP_OutputPixelFormatY4

4-bit monochrome pixels (1-plane Y luma, 4 bit truncation)

enumerator kPXP_OutputPixelFormatYUV2P422

16-bit pixels (2-plane UV interleaved bytes)

enumerator kPXP_OutputPixelFormatYUV2P420

16-bit pixels (2-plane UV)

enumerator kPXP__OutputPixelFormatYVU2P422
16-bit pixels (2-plane VU interleaved bytes)

enumerator kPXP__OutputPixelFormatYVU2P420
16-bit pixels (2-plane VU)

enum __pxp_ps_pixel_format
PXP process surface buffer pixel format.

Values:

enumerator kPXP__PsPixelFormatRGB888
32-bit pixels without alpha (unpacked 24-bit format)

enumerator kPXP__PsPixelFormatRGB555
16-bit pixels without alpha.

enumerator kPXP__PsPixelFormatRGB444
16-bit pixels without alpha.

enumerator kPXP__PsPixelFormatRGB565
16-bit pixels without alpha.

enumerator kPXP__PsPixelFormatYUV1P444
32-bit pixels (1-plane XYUV unpacked).

enumerator kPXP__PsPixelFormatUYVY1P422
16-bit pixels (1-plane U0,Y0,V0,Y1 interleaved bytes)

enumerator kPXP__PsPixelFormatVYUY1P422
16-bit pixels (1-plane V0,Y0,U0,Y1 interleaved bytes)

enumerator kPXP__PsPixelFormatY8
8-bit monochrome pixels (1-plane Y luma output)

enumerator kPXP__PsPixelFormatY4
4-bit monochrome pixels (1-plane Y luma, 4 bit truncation)

enumerator kPXP__PsPixelFormatYUV2P422
16-bit pixels (2-plane UV interleaved bytes)

enumerator kPXP__PsPixelFormatYUV2P420
16-bit pixels (2-plane UV)

enumerator kPXP__PsPixelFormatYVU2P422
16-bit pixels (2-plane VU interleaved bytes)

enumerator kPXP__PsPixelFormatYVU2P420
16-bit pixels (2-plane VU)

enumerator kPXP__PsPixelFormatYVU422
16-bit pixels (3-plane)

enumerator kPXP__PsPixelFormatYVU420
16-bit pixels (3-plane)

enum __pxp_ps_yuv_format
PXP process surface buffer YUV format.

Values:

enumerator kPXP__PsYUVFormatYUV
YUV format.

enumerator kPXP_PsYUVFormatYCbCr
YCbCr format.

enum _pxp_as_pixel_format

PXP alpha surface buffer pixel format.

Values:

enumerator kPXP_AsPixelFormatARGB8888
32-bit pixels with alpha.

enumerator kPXP_AsPixelFormatRGB888
32-bit pixels without alpha (unpacked 24-bit format)

enumerator kPXP_AsPixelFormatARGB1555
16-bit pixels with alpha.

enumerator kPXP_AsPixelFormatARGB4444
16-bit pixels with alpha.

enumerator kPXP_AsPixelFormatRGB555
16-bit pixels without alpha.

enumerator kPXP_AsPixelFormatRGB444
16-bit pixels without alpha.

enumerator kPXP_AsPixelFormatRGB565
16-bit pixels without alpha.

enum _pxp_alpha_mode

PXP alpha mode during blending.

Values:

enumerator kPXP_AlphaEmbedded
The alpha surface pixel alpha value will be used for blend.

enumerator kPXP_AlphaOverride
The user defined alpha value will be used for blend directly.

enumerator kPXP_AlphaMultiply
The alpha surface pixel alpha value scaled the user defined alpha value will be used for blend, for example, pixel alpha set to 200, user defined alpha set to 100, then the result alpha is $200 * 100 / 255$.

enumerator kPXP_AlphaRop
Raster operation.

enum _pxp_rop_mode

PXP ROP mode during blending.

Explanation:

- AS: Alpha surface
- PS: Process surface
- nAS: Alpha surface NOT value
- nPS: Process surface NOT value

Values:

enumerator kPXP_RopMaskAs
AS AND PS.

enumerator kPXP_RopMaskNotAs
nAS AND PS.

enumerator kPXP_RopMaskAsNot
AS AND nPS.

enumerator kPXP_RopMergeAs
AS OR PS.

enumerator kPXP_RopMergeNotAs
nAS OR PS.

enumerator kPXP_RopMergeAsNot
AS OR nPS.

enumerator kPXP_RopNotCopyAs
nAS.

enumerator kPXP_RopNot
nPS.

enumerator kPXP_RopNotMaskAs
AS NAND PS.

enumerator kPXP_RopNotMergeAs
AS NOR PS.

enumerator kPXP_RopXorAs
AS XOR PS.

enumerator kPXP_RopNotXorAs
AS XNOR PS.

enum _pxp_block_size
PXP process block size.

Values:

enumerator kPXP_BlockSize8
Process 8x8 pixel blocks.

enumerator kPXP_BlockSize16
Process 16x16 pixel blocks.

enum _pxp_csc1_mode
PXP CSC1 mode.

Values:

enumerator kPXP_Csc1YUV2RGB
YUV to RGB.

enumerator kPXP_Csc1YCbCr2RGB
YCbCr to RGB.

enum _pxp_csc2_mode
PXP CSC2 mode.

Values:

enumerator kPXP_Csc2YUV2RGB
YUV to RGB.

enumerator kPXP_Csc2YCbCr2RGB
YCbCr to RGB.

enumerator kPXP_Csc2RGB2YUV
RGB to YUV.

enumerator kPXP_Csc2RGB2YCbCr
RGB to YCbCr.

enum _pxp_ram
PXP internal memory.

Values:

enumerator kPXP_RamDither0Lut
Dither 0 LUT memory.

enumerator kPXP_RamDither1Lut
Dither 1 LUT memory.

enumerator kPXP_RamDither2Lut
Dither 2 LUT memory.

enumerator kPXP_RamDither0Err0
Dither 0 ERR0 memory.

enumerator kPXP_RamDither0Err1
Dither 0 ERR1 memory.

enumerator kPXP_RamAluA
ALU A instr memory.

enumerator kPXP_RamAluB
ALU B instr memory.

enumerator kPXP_WfeAFetch
WFE-A fetch memory.

enumerator kPXP_WfeBFetch
WFE-B fetch memory.

enum _pxp_dither_mode
PXP dither mode.

Values:

enumerator kPXP_DitherPassThrough
Pass through, no dither.

enumerator kPXP_DitherFloydSteinberg
Floyd-Steinberg. For dither engine 0 only.

enumerator kPXP_DitherAtkinson
Atkinson. For dither engine 0 only.

enumerator kPXP_DitherOrdered
Ordered dither.

enumerator kPXP_DitherQuantOnly
No dithering, only quantization.

enumerator kPXP_DitherSierra
Sierra. For dither engine 0 only.

enum _pxp_dither_lut_mode

PXP dither LUT mode.

Values:

enumerator kPXP_DitherLutOff

The LUT memory is not used for LUT, could be used as ordered dither index matrix.

enumerator kPXP_DitherLutPreDither

Use LUT at the pre-dither stage, The pre-dither LUT could only be used in Floyd mode or Atkinson mode, which are not supported by current PXP module.

enumerator kPXP_DitherLutPostDither

Use LUT at the post-dither stage.

enum _pxp_dither_matrix_size

PXP dither matrix size.

Values:

enumerator kPXP_DitherMatrix4

The dither index matrix is 4x4.

enumerator kPXP_DitherMatrix8

The dither index matrix is 8x8.

enumerator kPXP_DitherMatrix16

The dither index matrix is 16x16.

Porter Duff factor mode. .

Values:

enumerator kPXP_PorterDuffFactorOne

Use 1.

enumerator kPXP_PorterDuffFactorZero

Use 0.

enumerator kPXP_PorterDuffFactorStraight

Use straight alpha.

enumerator kPXP_PorterDuffFactorInversed

Use inversed alpha.

Porter Duff global alpha mode. .

Values:

enumerator kPXP_PorterDuffGlobalAlpha

Use global alpha.

enumerator kPXP_PorterDuffLocalAlpha

Use local alpha in each pixel.

enumerator kPXP_PorterDuffScaledAlpha

Use global alpha * local alpha.

Porter Duff alpha mode. .

Values:

enumerator kPXP_PorterDuffAlphaStraight
Use straight alpha, $s0\_alpha' = s0\_alpha$.

enumerator kPXP_PorterDuffAlphaInversed
Use inversed alpha, $s0\_alpha' = 0xFF - s0\_alpha$.

Porter Duff color mode. .

Values:

enumerator kPXP_PorterDuffColorStraight

Deprecated:

Use kPXP_PorterDuffColorNoAlpha.

enumerator kPXP_PorterDuffColorInversed

Deprecated:

Use kPXP_PorterDuffColorWithAlpha.

enumerator kPXP_PorterDuffColorNoAlpha
 $s0\_pixel' = s0\_pixel$.

enumerator kPXP_PorterDuffColorWithAlpha
 $s0\_pixel' = s0\_pixel * s0\_alpha$ ".

enum _pxp_porter_duff_blend_mode

PXP Porter Duff blend mode. Note: don't change the enum item value.

Values:

enumerator kPXP_PorterDuffSrc
Source Only

enumerator kPXP_PorterDuffAtop
Source Atop

enumerator kPXP_PorterDuffOver
Source Over

enumerator kPXP_PorterDuffIn
Source In.

enumerator kPXP_PorterDuffOut
Source Out.

enumerator kPXP_PorterDuffDst
Destination Only.

enumerator kPXP_PorterDuffDstAtop
Destination Atop.

enumerator kPXP_PorterDuffDstOver
Destination Over.

enumerator kPXP_PorterDuffDstIn
Destination In.

enumerator kPXP_PorterDuffDstOut
Destination Out.

enumerator kPXP_PorterDuffXor
XOR.

enumerator kPXP__PorterDuffClear
Clear.

enumerator kPXP__PorterDuffMax

enum __pxp__process__engine__name
PXP process engine enumeration.

Values:

enumerator kPXP__PsAsOutEngine

enumerator kPXP__DitherEngine

enumerator kPXP__WfeaEngine

enumerator kPXP__WfebEngine

enumerator kPXP__InputFetchStoreEngine

enumerator kPXP__Alpha1Engine

enumerator kPXP__Csc2Engine

enumerator kPXP__LutEngine

enumerator kPXP__Rotate0Engine

enumerator kPXP__Rotate1Engine

enum __pxp__fetch__engine__name
PXP fetch engine enumeration.

There are actually 4 fetch engine implemented, the others are WFE-A fetch engine and WFE-B fetch engine, whose registers are reserved from developer.

Values:

enumerator kPXP__FetchInput

enumerator kPXP__FetchDither

enum __pxp__fetch__interface__mode
PXP fetch engine interface mode with the upstream store engine.

Values:

enumerator kPXP__FetchModeNormal

enumerator kPXP__FetchModeHandshake

enumerator kPXP__FetchModeBypass

enum __pxp__scanline__burst
PXP fetch/store engine burst length for scanline mode.

Values:

enumerator kPXP__Scanline8bytes

enumerator kPXP__Scanline16bytes

enumerator kPXP__Scanline32bytes

enumerator kPXP__Scanline64bytes

enum `_pxp_activeBits`

PXP fetch/store engine input/output active bits configuration.

Since fetch engine is 64-bit input and 32-bit output per channel, need to configure both channels to use 64-bit input mode. And expand configuration will have no effect.

Values:

enumerator `kPXP_Active8Bits`

enumerator `kPXP_Active16Bits`

enumerator `kPXP_Active32Bits`

enumerator `kPXP_Active64Bits`

enum `_pxp_fetch_output_word_order`

PXP fetch engine output word order when using 2 channels for 64-bit mode.

Values:

enumerator `kPXP_FetchOutputChannel1channel0`

In 64bit mode, channel 1 output high byte.

enumerator `kPXP_FetchOutputChannel0channel1`

In 64bit mode, channel 0 output high byte.

enum `_pxp_fetch_pixel_format`

PXP fetch engine input pixel format.

Values:

enumerator `kPXP_FetchFormatRGB565`

enumerator `kPXP_FetchFormatRGB555`

enumerator `kPXP_FetchFormatARGB1555`

enumerator `kPXP_FetchFormatRGB444`

enumerator `kPXP_FetchFormatARGB4444`

enumerator `kPXP_FetchFormatYUYVOrYVYU`

enumerator `kPXP_FetchFormatUYVYOrVYUY`

enumerator `kPXP_FetchFormatYUV422_2P`

enum `_pxp_store_engine_name`

PXP store engine enumeration.

There are actually 4 store engine implemented, the others are WFE-A store engine and WFE-B store engine, whose registers are reserved from developer.

Values:

enumerator `kPXP_StoreInput`

enumerator `kPXP_StoreDither`

enum `_pxp_store_interface_mode`

PXP store engine interface mode with the downstream fetch engine.

Values:

enumerator kPXP_StoreModeBypass

Store engine output the input data, after the shift function directly to the downstream Fetch Engine.

enumerator kPXP_StoreModeNormal

Store engine stores the input data to memory.

enumerator kPXP_StoreModeHandshake

Downstream fetch engine fetch data per scanline from memory using buffer sharing with store engine.

enumerator kPXP_StoreModeDual

Store engine outputs data directly to downstream fetch engine(Bypass) but also storing it to memory at the same time.

enum __pxp_store_yuv_mode

PXP store engine YUV output mode.

Values:

enumerator kPXP_StoreYUVDisable

Do not output YUV pixel format.

enumerator kPXP_StoreYUVPlane1

Use channel to output YUV422_1p pixel format, need to use shift operation to make sure each pixel component in its proper position: 64-bits of pixel data format and each 32 bits as {Y0, U0, Y1, V0}.

enumerator kPXP_StoreYUVPlane2

Use channel to output YUV422_2p pixel format, need to use shift operation to make sure each pixel component in its proper position: channel 0 {Y0,Y1}, channel 1 {U0,V0}.

enum __pxp_cfa_input_format

PXP pre-dither CFA engine input pixel format.

Values:

enumerator kPXP_CfaRGB888

enumerator kPXP_CfaRGB444

enum __pxp_histogram_mask_condition

PXP histogram mask condition.

Values:

enumerator kPXP_HistogramMaskEqual

Value that equal to value0 will pass the mask operation.

enumerator kPXP_HistogramMaskNotEqual

Value that not equal to value0 will pass the mask operation.

enumerator kPXP_HistogramMaskIn

Value that within the range of value0-value1 will pass the mask operation.

enumerator kPXP_HistogramMaskOut

Value that without the range of value0-value1 will pass the mask operation.

enum __pxp_histogram_flags

PXP Histogram operation result flags.

Values:

enumerator kPXP_Histogram2levelMatch

enumerator kPXP__Histogram4levelMatch

enumerator kPXP__Histogram8levelMatch

enumerator kPXP__Histogram16levelMatch

enumerator kPXP__Histogram32levelMatch

typedef enum *_pxp_flip_mode* pxp_flip_mode_t
PXP output flip mode.

typedef enum *_pxp_rotate_position* pxp_rotate_position_t
PXP rotate mode.

typedef enum *_pxp_rotate_degree* pxp_rotate_degree_t
PXP rotate degree.

typedef enum *_pxp_interlaced_output_mode* pxp_interlaced_output_mode_t
PXP interlaced output mode.

typedef enum *_pxp_output_pixel_format* pxp_output_pixel_format_t
PXP output buffer format.

typedef struct *_pxp_output_buffer_config* pxp_output_buffer_config_t
PXP output buffer configuration.

typedef enum *_pxp_ps_pixel_format* pxp_ps_pixel_format_t
PXP process surface buffer pixel format.

typedef enum *_pxp_ps_yuv_format* pxp_ps_yuv_format_t
PXP process surface buffer YUV format.

typedef struct *_pxp_ps_buffer_config* pxp_ps_buffer_config_t
PXP process surface buffer configuration.

typedef enum *_pxp_as_pixel_format* pxp_as_pixel_format_t
PXP alpha surface buffer pixel format.

typedef struct *_pxp_as_buffer_config* pxp_as_buffer_config_t
PXP alphas surface buffer configuration.

typedef enum *_pxp_alpha_mode* pxp_alpha_mode_t
PXP alpha mode during blending.

typedef enum *_pxp_rop_mode* pxp_rop_mode_t
PXP ROP mode during blending.

Explanation:

- AS: Alpha surface
- PS: Process surface
- nAS: Alpha surface NOT value
- nPS: Process surface NOT value

typedef struct *_pxp_as_blend_config* pxp_as_blend_config_t
PXP alpha surface blending configuration.

typedef struct *_pxp_as_blend_secondary_config* pxp_as_blend_secondary_config_t
PXP secondary alpha surface blending engine configuration.

typedef enum *_pxp_block_size* pxp_block_size_t
PXP process block size.

```
typedef enum _pxp_csc1_mode pxp_csc1_mode_t
```

 PXP CSC1 mode.

```
typedef enum _pxp_csc2_mode pxp_csc2_mode_t
```

 PXP CSC2 mode.

```
typedef struct _pxp_csc2_config pxp_csc2_config_t
```

 PXP CSC2 configuration.

Converting from YUV/YCbCr color spaces to the RGB color space uses the following equation structure:

$$R = A1(Y+D1) + A2(U+D2) + A3(V+D3) \quad G = B1(Y+D1) + B2(U+D2) + B3(V+D3) \quad B = C1(Y+D1) + C2(U+D2) + C3(V+D3)$$

Converting from the RGB color space to YUV/YCbCr color spaces uses the following equation structure:

$$Y = A1*R + A2*G + A3*B + D1 \quad U = B1*R + B2*G + B3*B + D2 \quad V = C1*R + C2*G + C3*B + D3$$

```
typedef enum _pxp_ram pxp_ram_t
```

 PXP internal memory.

```
typedef struct _pxp_dither_final_lut_data pxp_dither_final_lut_data_t
```

 PXP dither final LUT data.

```
typedef struct _pxp_dither_config pxp_dither_config_t
```

 PXP dither configuration.

```
typedef enum _pxp_porter_duff_blend_mode pxp_porter_duff_blend_mode_t
```

 PXP Porter Duff blend mode. Note: don't change the enum item value.

```
typedef struct _pxp_pic_copy_config pxp_pic_copy_config_t
```

 PXP Porter Duff blend mode. Note: don't change the enum item value.

```
typedef enum _pxp_process_engine_name pxp_process_engine_name_t
```

 PXP process engine enumeration.

```
typedef enum _pxp_fetch_engine_name pxp_fetch_engine_name_t
```

 PXP fetch engine enumeration.

There are actually 4 fetch engine implemented, the others are WFE-A fetch engine and WFE-B fetch engine, whose registers are reserved from developer.

```
typedef enum _pxp_fetch_interface_mode pxp_fetch_interface_mode_t
```

 PXP fetch engine interface mode with the upstream store engine.

```
typedef enum _pxp_scanline_burst pxp_scanline_burst_t
```

 PXP fetch/store engine burst length for scanline mode.

```
typedef struct _pxp_block_format_config pxp_block_config_t
```

 PXP fetch engine block configuration.

```
typedef enum _pxp_activeBits pxp_active_bits_t
```

 PXP fetch/store engine input/output active bits configuration.

Since fetch engine is 64-bit input and 32-bit output per channel, need to configure both channels to use 64-bit input mode. And expand configuration will have no effect.

```
typedef enum _pxp_fetch_output_word_order pxp_fetch_output_word_order_t
```

 PXP fetch engine output word order when using 2 channels for 64-bit mode.

`typedef struct _pxp_fetch_shift_component pxp_fetch_shift_component_t`
PXP fetch engine shift component configuration.

Fetch engine can divided each word into 4 components and shift them.

`typedef struct _pxp_fetch_shift_config pxp_fetch_shift_config_t`
PXP fetch engine shift configuration.

Fetch engine can divided each word into 4 components and shift them. For example, to change YUV444 to YVU444, U and V positions need to be shifted: OFFSET0=8, OFFSET1=0, OFFSET2=16, OFFSET3=24, WIDTH0/1/2/3=8

`typedef enum _pxp_fetch_pixel_format pxp_fetch_pixel_format_t`
PXP fetch engine input pixel format.

`typedef struct _pxp_fetch_engine_config pxp_fetch_engine_config_t`
PXP fetch engine configuration for one of the channel.

`typedef enum _pxp_store_engine_name pxp_store_engine_name_t`
PXP store engine enumeration.

There are actually 4 store engine implemented, the others are WFE-A store engine and WFE-B store engine, whose registers are reserved from developer.

`typedef enum _pxp_store_interface_mode pxp_store_interface_mode_t`
PXP store engine interface mode with the downstream fetch engine.

`typedef enum _pxp_store_yuv_mode pxp_store_yuv_mode_t`
PXP store engine YUV output mode.

`typedef struct _pxp_store_shift_config pxp_store_shift_config_t`
Shift configuration for PXP store engine.

`typedef struct _pxp_store_engine_config pxp_store_engine_config_t`
PXP store engine configuration for one of the channel.

`typedef enum _pxp_cfa_input_format pxp_cfa_input_format_t`
PXP pre-dither CFA engine input pixel format.

`typedef struct _pxp_cfa_config pxp_cfa_config_t`
PXP pre-dither CFA engine configuration.

`typedef enum _pxp_histogram_mask_condition pxp_histogram_mask_condition_t`
PXP histogram mask condition.

`typedef struct _pxp_histogram_config pxp_histogram_config_t`
PXP Histogram configuration.

`typedef struct _pxp_histogram_mask_result pxp_histogram_mask_result_t`
PXP Histogram mask result.

`typedef struct _pxp_wfea_engine_config pxp_wfea_engine_config_t`
PXP WFE-A engine configuration.

PXP_LUT_TABLE_BYTE

PXP_INTERNAL_RAM_LUT_BYTE

PXP_SHARE_ROTATE

PXP_USE_PATH

PXP_COMBINE_BYTE_TO_WORD(dataAddr)

`struct _pxp_output_buffer_config`
`#include <fsl_pxp.h>` PXP output buffer configuration.

Public Members

pxp_output_pixel_format_t pixelFormat

Output buffer pixel format.

pxp_interlaced_output_mode_t interlacedMode

Interlaced output mode.

uint32_t buffer0Addr

Output buffer 0 address.

uint32_t buffer1Addr

Output buffer 1 address, used for UV data in YUV 2-plane mode, or field 1 in output interlaced mode.

uint16_t pitchBytes

Number of bytes between two vertically adjacent pixels.

uint16_t width

Pixels per line.

uint16_t height

How many lines in output buffer.

struct __pxp_ps_buffer_config

#include <fsl_pxp.h> PXP process surface buffer configuration.

Public Members

pxp_ps_pixel_format_t pixelFormat

PS buffer pixel format.

bool swapByte

For each 16 bit word, set true to swap the two bytes.

uint32_t bufferAddr

Input buffer address for the first panel.

uint32_t bufferAddrU

Input buffer address for the second panel.

uint32_t bufferAddrV

Input buffer address for the third panel.

uint16_t pitchBytes

Number of bytes between two vertically adjacent pixels.

struct __pxp_as_buffer_config

#include <fsl_pxp.h> PXP alphas surface buffer configuration.

Public Members

pxp_as_pixel_format_t pixelFormat

AS buffer pixel format.

uint32_t bufferAddr

Input buffer address.

uint16_t pitchBytes

Number of bytes between two vertically adjacent pixels.

struct __pxp_as_blend_config
#include <fsl_pxp.h> PXP alpha surface blending configuration.

Public Members

uint8_t alpha
User defined alpha value, only used when alphaMode is kPXP_AlphaOverride or kPXP_AlphaRop.

bool invertAlpha
Set true to invert the alpha.

pxp_alpha_mode_t alphaMode
Alpha mode.

pxp_rop_mode_t ropMode
ROP mode, only valid when alphaMode is kPXP_AlphaRop.

struct __pxp_as_blend_secondary_config
#include <fsl_pxp.h> PXP secondary alpha surface blending engine configuration.

Public Members

bool invertAlpha
Set true to invert the alpha.

bool ropEnable
Enable rop mode.

pxp_rop_mode_t ropMode
ROP mode, only valid when ropEnable is true.

struct __pxp_csc2_config
#include <fsl_pxp.h> PXP CSC2 configuration.

Converting from YUV/YCbCr color spaces to the RGB color space uses the following equation structure:

$$R = A1(Y+D1) + A2(U+D2) + A3(V+D3) \quad G = B1(Y+D1) + B2(U+D2) + B3(V+D3) \quad B = C1(Y+D1) + C2(U+D2) + C3(V+D3)$$

Converting from the RGB color space to YUV/YCbCr color spaces uses the following equation structure:

$$Y = A1*R + A2*G + A3*B + D1 \quad U = B1*R + B2*G + B3*B + D2 \quad V = C1*R + C2*G + C3*B + D3$$

Public Members

pxp_csc2_mode_t mode
Conversion mode.

float A1
A1.

float A2
A2.

float A3
A3.

float B1
B1.

float B2
B2.

float B3
B3.

float C1
C1.

float C2
C2.

float C3
C3.

int16_t D1
D1.

int16_t D2
D2.

int16_t D3
D3.

struct _pxp_dither_final_lut_data
#include <fsl_pxp.h> PXP dither final LUT data.

Public Members

uint32_t data_3_0
Data 3 to data 0. Data 0 is the least significant byte.

uint32_t data_7_4
Data 7 to data 4. Data 4 is the least significant byte.

uint32_t data_11_8
Data 11 to data 8. Data 8 is the least significant byte.

uint32_t data_15_12
Data 15 to data 12. Data 12 is the least significant byte.

struct _pxp_dither_config
#include <fsl_pxp.h> PXP dither configuration.

Public Members

uint32_t enableDither0
Enable dither engine 0 or not, set 1 to enable, 0 to disable.

uint32_t enableDither1
Enable dither engine 1 or not, set 1 to enable, 0 to disable.

uint32_t enableDither2
Enable dither engine 2 or not, set 1 to enable, 0 to disable.

uint32_t ditherMode0
Dither mode for dither engine 0. See _pxp_dither_mode.

uint32_t ditherMode1

Dither mode for dither engine 1. See `_pxp_dither_mode`.

uint32_t ditherMode2

Dither mode for dither engine 2. See `_pxp_dither_mode`.

uint32_t quantBitNum

Number of bits quantize down to, the valid value is 1~7.

uint32_t lutMode

How to use the memory LUT, see `_pxp_dither_lut_mode`. This must be set to `kPXP_DitherLutOff` if any dither engine uses `kPXP_DitherOrdered` mode.

uint32_t idxMatrixSize0

Size of index matrix used for dither for dither engine 0, see `_pxp_dither_matrix_size`.

uint32_t idxMatrixSize1

Size of index matrix used for dither for dither engine 1, see `_pxp_dither_matrix_size`.

uint32_t idxMatrixSize2

Size of index matrix used for dither for dither engine 2, see `_pxp_dither_matrix_size`.

uint32_t enableFinalLut

Enable the final LUT, set 1 to enable, 0 to disable.

struct pxp_porter_duff_config_t

#include <fsl_pxp.h> PXP Porter Duff configuration.

Public Members

uint32_t enable

Enable or disable Porter Duff.

uint32_t srcFactorMode

Source layer (or AS, s1) factor mode, see `pxp_porter_duff_factor_mode`.

uint32_t dstGlobalAlphaMode

Destination layer (or PS, s0) global alpha mode, see `pxp_porter_duff_global_alpha_mode`.

uint32_t dstAlphaMode

Destination layer (or PS, s0) alpha mode, see `pxp_porter_duff_alpha_mode`.

uint32_t dstColorMode

Destination layer (or PS, s0) color mode, see `pxp_porter_duff_color_mode`.

uint32_t dstFactorMode

Destination layer (or PS, s0) factor mode, see `pxp_porter_duff_factor_mode`.

uint32_t srcGlobalAlphaMode

Source layer (or AS, s1) global alpha mode, see `pxp_porter_duff_global_alpha_mode`.

uint32_t srcAlphaMode

Source layer (or AS, s1) alpha mode, see `pxp_porter_duff_alpha_mode`.

uint32_t srcColorMode

Source layer (or AS, s1) color mode, see `pxp_porter_duff_color_mode`.

uint32_t dstGlobalAlpha

Destination layer (or PS, s0) global alpha value, 0~255.

uint32_t srcGlobalAlpha

Source layer (or AS, s1) global alpha value, 0~255.

struct __pxp_pic_copy_config

#include <fsl_pxp.h> PXP Porter Duff blend mode. Note: don't change the enum item value.

Public Members

uint32_t srcPicBaseAddr

Source picture base address.

uint16_t srcPitchBytes

Pitch of the source buffer.

uint16_t srcOffsetX

Copy position in source picture.

uint16_t srcOffsetY

Copy position in source picture.

uint32_t destPicBaseAddr

Destination picture base address.

uint16_t destPitchBytes

Pitch of the destination buffer.

uint16_t destOffsetX

Copy position in destination picture.

uint16_t destOffsetY

Copy position in destination picture.

uint16_t width

Pixel number each line to copy.

uint16_t height

Lines to copy.

pxp_as_pixel_format_t pixelFormat

Buffer pixel format.

struct __pxp_block_format_config

#include <fsl_pxp.h> PXP fetch engine block configuration.

Public Members

bool enableblock

Enable to use block mode instead of scanline mode. Note: 1.Make sure to enable if rotate or flip mode is enabled. 2.Block mode cannot work on 64bpp data stream where activeBits = 64. 3. If LUT processing is in the path between the fetch and store engind, block mode must be enabled.

bool blockSize16

Enable to use 16*16 block, otherwise it will be 8*8 block.

pxp_scanline_burst_t burstLength

When using scanline mode, configure this for burst length.

struct __pxp_fetch_shift_component

#include <fsl_pxp.h> PXP fetch engine shift component configuration.

Fetch engine can divided each word into 4 components and shift them.

struct __pxp_fetch_shift_config

#include <fsl_pxp.h> PXP fetch engine shift configuration.

Fetch engine can divided each word into 4 components and shift them. For example, to change YUV444 to YVU444, U and V positions need to be shifted: OFFSET0=8, OFFSET1=0, OFFSET2=16, OFFSET3=24, WIDTH0/1/2/3=8

struct __pxp_fetch_engine_config

#include <fsl_pxp.h> PXP fetch engine configuration for one of the channel.

Public Members

bool channelEnable

Enable channel.

uint32_t inputBaseAddr0

The input base address. Used for Y plane input when pixel format is YUV422_2p.

uint32_t inputBaseAddr1

Must configure this for UV plane when input pixel format is YUV422_2p.

uint16_t totalHeight

Total height for the actual fetch size.

uint16_t totalWidth

Total width for the actual fetch size.

uint16_t pitchBytes

Channel input pitch

uint16_t ulcX

X coordinate of upper left coordinate in pixels of the active area of the total input memory

uint16_t ulcY

Y coordinate of upper left coordinate in pixels of the active area of the total input memory

uint16_t lrcX

X coordinate of Lower right coordinate in pixels of the active area of the total input memory

uint16_t lrcY

Y coordinate of Lower right coordinate in pixels of the active area of the total input memory

uint32_t backGroundColor

Pixel value of the background color for the space outside the active area.

pxp_fetch_interface_mode_t interface

Interface mode, normal/bypass/handshake

pxp_active_bits_t activeBits

Input active bits.

pxp_fetch_pixel_format_t pixelFormat

Input pixel fetch format

bool expandEnable

If enabled, input pixel will be expanded to ARGB8888, RGB888 or YUV444 of 32-bit format at the output.

pxp_flip_mode_t flipMode

Flip the fetched input.

pxp_rotate_degree_t rotateDegree

Rotate the fetched input.

pxp_block_config_t fetchFormat

Block mode configuration. Make sure to enable block if rotate or flip mode is enabled.

pxp_fetch_shift_config_t shiftConfig

Shift operation configuration.

pxp_fetch_output_word_order_t wordOrder

Output word order when using 2 channels for 64-bit mode.

struct __pxp_store_shift_config

#include <fsl_pxp.h> Shift configuration for PXP store engine.

Public Members

bool shiftBypass

Bypass the data shift

uint64_t *pDataShiftMask

Pointer to mask0~mask7 to mask the 64-bit of output data, data is masked first then shifted according to width.

uint8_t *pDataShiftWidth

Pointer to width0~width7. Bit 7 is for shifted direction, 0 to right. Bit0~5 is for shift width.

uint8_t *pFlagShiftMask

Pointer to mask0~mask7 to mask the 8-bit of output flag, flag is masked first then shifted according to width.

uint8_t *pFlagShiftWidth

Pointer to width0~width7. Bit 6 is for shifted direction, 0 to right. Bit0~5 is for shift width.

struct __pxp_store_engine_config

#include <fsl_pxp.h> PXP store engine configuration for one of the channel.

Public Members

bool channelEnable

Enable channel.

uint32_t outputBaseAddr0

The channel 0 output address if using 2 channels. If using 1 channel(must be channel 0) and YUV422_2p output format, is for Y plane address.

uint32_t outputBaseAddr1

The channel 1 output address if using 2 channels. If using 1 channel(must be channel 0) and YUV422_2p output format, is for UV plane address.

uint16_t totalHeight

Total height for the actual store size.

uint16_t totalWidth

Total width for the actual store size.

uint16_t pitchBytes

Channel input pitch

pxp_store_interface_mode_t interface

Interface mode, normal/bypass/handshake/dual. Make sure 2 channels use the same mode if both enabled.

pxp_active_bits_t activeBits

Output active bits.

pxp_store_yuv_mode_t yuvMode

Whether to output YUV pixel format.

bool useFixedData

Whether to use fixed value for the output data. Can be used to write fixed value to specific memory location for memory initialization.

uint32_t fixedData

The value of the fixed data.

bool packInSelect

When enabled, channel 0 will select low 32 bit shift out data to pack while channel i select high 32 bit, otherwise all 64-bit of data will be selected.

pxp_block_config_t storeFormat

The format to store data, block or otherwise.

pxp_store_shift_config_t shiftConfig

Shift operation configuration.

struct _pxp_cfa_config

#include <fsl_pxp.h> PXP pre-dither CFA engine configuration.

Public Members

bool bypass

Bypass the CFA process

pxp_cfa_input_format_t pixelInFormat

The pixel input format for CFA.

uint8_t arrayWidth

CFA array vertical size in pixels, min 3 max 15.

uint8_t arrayHeight

CFA array horizontal size in pixels, min 3 max 15.

uint16_t totalHeight

Total height for the buffer size, make sure it is aligned with the dither fetch engine and dither engine.

uint16_t totalWidth

Total width for the buffer size, make sure it is aligned with the dither fetch engine and dither engine.

uint32_t *cfaValue

Pointer to the value for the CFA array. 2-bit per component: 00-R,01-G,10-B,11-W. For a 4x4 array, 32 bits are need.

struct __pxp_histogram_config

#include <fsl_pxp.h> PXP Histogram configuration.

Public Members

bool enable

Enable histogram process.

uint8_t *pParamValue

Pointer to the 62(2+4+8+16+32) byte of param value for 2-level, 4-level.....32-level parameters. Only low 5-bit of each byte is valid.

uint8_t lutValueOffset

The starting bit position of the LUT value.

uint8_t lutValueWidth

The bit width of the LUT value, should be no more than 6 bits since only 63 LUTs are supported.

bool enableMask

Enable mask operation.

uint8_t maskValue0

Value 0 for the condition judgement.

uint8_t maskValue1

Value 1 for the condition judgement.

uint8_t maskOffset

The starting bit position of the field to be checked against mask condition.

uint8_t maskWidth

The width of the field to be checked against mask condition.

pxp_histogram_mask_condition_t condition

The mask condition.

uint16_t totalHeight

Total height for the buffer size, make sure it is aligned with the output of legacy flow or the WFE-A/B engine.

uint16_t totalWidth

Total width for the buffer size, make sure it is aligned with the output of legacy flow or the WFE-A/B engine.

struct __pxp_histogram_mask_result

#include <fsl_pxp.h> PXP Histogram mask result.

Public Members

uint32_t pixelCount

The total count of the pixels that pass the mask(collided pixels).

uint32_t minX

The x offset of the ULC of the minimal histogram that covers all passed pixels.

uint32_t minY

The y offset of the ULC of the minimal histogram that covers all passed pixels.

uint32_t maxX

The x offset of the LRC of the minimal histogram that covers all passed pixels.

uint32_t maxY

The y offset of the LRC of the minimal histogram that covers all passed pixels.

uint64_t lutlist

The 64-bit LUT list of collided pixels, if pixel of LUT17 is collided, bit17 in the list is set.

struct __pxp_wfea_engine_config

#include <fsl_pxp.h> PXP WFE-A engine configuration.

Public Members

uint32_t y4Addr

Address for Y4 buffer.

uint32_t y4cAddr

Address for Y4C buffer, {Y4[3:0],3'b000,collision}, 8bpp.

uint32_t wbAddr

Address for EPDC working buffer.

uint16_t updateWidth

Width of the update area.

uint16_t updateHeight

Height of the update area.

uint16_t updatePitch

Pitch of the update area.

uint16_t ulcX

X coordinate of upper left coordinate of the total input memory

uint16_t ulcY

Y coordinate of upper left coordinate of the total input memory

uint16_t resX

Horizontal resolution in pixels.

uint8_t lutNum

The EPDC LUT number for the update.

bool fullUpdateEnable

Enable full update.

bool alphaEnable

Enable alpha field, upd is {Y4[3:0],3'b000,alpha} format, otherwise its {Y4[3:0],4'b0000}.

bool detectionOnly

Detection only, do not write working buffer.

2.61 Reset

enum __pcc_reset

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control in PCC registers

Values:

enumerator kRESET_IpInvalid

enumerator kRESET_Wdog0

enumerator kRESET_Wdog1

enumerator kRESET_Flexspi0

enumerator kRESET_Lpit0

enumerator kRESET_Flexio0

enumerator kRESET_I3c0

enumerator kRESET_Lpspi0

enumerator kRESET_Lpspi1

enumerator kRESET_Adc0

enumerator kRESET_Dac0

enumerator kRESET_Dac1

enumerator kRESET_Flexspi1

enumerator kRESET_Tpm0

enumerator kRESET_Tpm1

enumerator kRESET_Lpi2c0

enumerator kRESET_Lpi2c1

enumerator kRESET_Lpuart0

enumerator kRESET_Lpuart1

enumerator kRESET_Sai0

enumerator kRESET_Sai1

enumerator kRESET_Adc1

enumerator kRESET_Flexcan

enumerator kRESET_Wdog2

enumerator kRESET_Tpm2

enumerator kRESET_Tpm3

enumerator kRESET_Mrt

enumerator kRESET_Lpi2c2

enumerator kRESET_Lpi2c3
enumerator kRESET_I3c1
enumerator kRESET_Lpuart2
enumerator kRESET_Lpuart3
enumerator kRESET_Lpspi2
enumerator kRESET_Lpspi3
enumerator kRESET_Sai2
enumerator kRESET_Sai3
enumerator kRESET_Micfl
enumerator kRESET_Wdog3
enumerator kRESET_Wdog4
enumerator kRESET_Lpit1
enumerator kRESET_Tpm4
enumerator kRESET_Tpm5
enumerator kRESET_Flexio1
enumerator kRESET_I3c2
enumerator kRESET_Lpi2c4
enumerator kRESET_Lpi2c5
enumerator kRESET_Lpuart4
enumerator kRESET_Lpuart5
enumerator kRESET_Lpspi4
enumerator kRESET_Lpspi5
enumerator kRESET_Flexspi2
enumerator kRESET_Tpm6
enumerator kRESET_Tpm7
enumerator kRESET_Lpi2c6
enumerator kRESET_Lpi2c7
enumerator kRESET_Lpuart6
enumerator kRESET_Lpuart7
enumerator kRESET_Sai4
enumerator kRESET_Sai5
enumerator kRESET_Usdhc0
enumerator kRESET_Usdhc1

```
enumerator kRESET__Usdhc2
enumerator kRESET__Usb0
enumerator kRESET__Usb0Phy
enumerator kRESET__Usb1
enumerator kRESET__Usb1Phy
enumerator kRESET__Enet
enumerator kRESET__Tpm8
enumerator kRESET__Sai6
enumerator kRESET__Sai7
enumerator kRESET__Spdif
enumerator kRESET__Isi
enumerator kRESET__Csi
enumerator kRESET__Dsi
enumerator kRESET__Wdog5
enumerator kRESET__Epdc
enumerator kRESET__Pxp
enumerator kRESET__Gpu2d
enumerator kRESET__Gpu3d
enumerator kRESET__Dcnano
enumerator kRESET__Lpddr4
enumerator kRESET__CsiClkUi
```

```
typedef enum _pcc_reset pcc_reset_t
```

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control in PCC registers

```
typedef pcc_reset_t reset_ip_name_t
```

IP reset handle.

```
void RESET_SetPeripheralReset(reset_ip_name_t peripheral)
```

Assert reset to peripheral.

Asserts reset signal to specified peripheral module.

Parameters

- peripheral – Assert reset to this peripheral.

```
void RESET_ClearPeripheralReset(reset_ip_name_t peripheral)
```

Clear reset to peripheral.

Clears reset signal to specified peripheral module, allows it to operate.

Parameters

- peripheral – Clear reset to this peripheral.

void RESET_PeripheralReset(*reset_ip_name_t* peripheral)

Reset peripheral module.

Reset peripheral module.

Parameters

- peripheral – Peripheral to reset.

FSL_RESET_DRIVER_VERSION

reset driver version 2.0.0.

DAC_RSTS

Array initializers with peripheral reset bits

ENET_RSTS

EPDC_RSTS

FLEXCAN_RSTS

FLEXIO_RSTS

FLEXSPI_RSTS

I3C_RSTS

ISI_RSTS

LCDIF_RSTS

LPADC_RSTS

LPI2C_RSTS

LPIT_RSTS

LPSPI_RSTS

LPUART_RSTS

MIPI_DSI_RSTS

MRT_RSTS

PXP_RSTS

SAI_RSTS

SPDIF_RSTS

TPM_RSTS

USDHC_RSTS

WDOG_RSTS

2.62 RGPIO: Rapid General-Purpose Input/Output Driver

FSL_RGPIO_DRIVER_VERSION

RGPIO driver version 2.1.0.

enum _rgpio_pin_direction

RGPIO direction definition.

Values:

enumerator kRGPIO_DigitalInput

Set current pin as digital input

enumerator kRGPIO_DigitalOutput

Set current pin as digital output

enum _rgpio_checker_attribute

RGPIO checker attribute.

Values:

enumerator kRGPIO_UsernonsecureRWUsersecureRWPrivilegedsecureRW

User nonsecure:Read+Write; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureRUsersecureRWPrivilegedsecureRW

User nonsecure:Read; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureNUsersecureRWPrivilegedsecureRW

User nonsecure:None; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureRUsersecureRPrivilegedsecureRW

User nonsecure:Read; User Secure:Read; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureNUsersecureRPrivilegedsecureRW

User nonsecure:None; User Secure:Read; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureNUsersecureNPrivilegedsecureRW

User nonsecure:None; User Secure:None; Privileged Secure:Read+Write

enumerator kRGPIO_UsernonsecureNUsersecureNPrivilegedsecureR

User nonsecure:None; User Secure:None; Privileged Secure:Read

enumerator kRGPIO_UsernonsecureNUsersecureNPrivilegedsecureN

User nonsecure:None; User Secure:None; Privileged Secure:None

enumerator kRGPIO_IgnoreAttributeCheck

Ignores the attribute check

enum _rgpio_interrupt_sel

Configures the interrupt generation condition.

Values:

enumerator kRGPIO_InterruptOutput0

Interrupt/DMA request/trigger output 0.

enumerator kRGPIO_InterruptOutput1

Interrupt/DMA request/trigger output 1.

enumerator kRGPIO_InterruptOutput2

Interrupt/DMA request/trigger output 2.

enumerator kRGPIO_InterruptOutput3

Interrupt/DMA request/trigger output 3.

enum _rgpio_interrupt_config

Configures the interrupt generation condition.

Values:

enumerator `kRGPIO_InterruptOrDMADisabled`

Interrupt/DMA request is disabled.

enumerator `kRGPIO_DMARisingEdge`

DMA request on rising edge.

enumerator `kRGPIO_DMAFallingEdge`

DMA request on falling edge.

enumerator `kRGPIO_DMAEitherEdge`

DMA request on either edge.

enumerator `kRGPIO_FlagRisingEdge`

Flag sets on rising edge.

enumerator `kRGPIO_FlagFallingEdge`

Flag sets on falling edge.

enumerator `kRGPIO_FlagEitherEdge`

Flag sets on either edge.

enumerator `kRGPIO_InterruptLogicZero`

Interrupt when logic zero.

enumerator `kRGPIO_InterruptRisingEdge`

Interrupt on rising edge.

enumerator `kRGPIO_InterruptFallingEdge`

Interrupt on falling edge.

enumerator `kRGPIO_InterruptEitherEdge`

Interrupt on either edge.

enumerator `kRGPIO_InterruptLogicOne`

Interrupt when logic one.

enumerator `kRGPIO_ActiveHighTriggerOutputEnable`

Enable active high-trigger output.

enumerator `kRGPIO_ActiveLowTriggerOutputEnable`

Enable active low-trigger output.

typedef enum *rgpio_pin_direction* `rgpio_pin_direction_t`

RGPIO direction definition.

typedef enum *rgpio_checker_attribute* `rgpio_checker_attribute_t`

RGPIO checker attribute.

typedef enum *rgpio_interrupt_sel* `rgpio_interrupt_sel_t`

Configures the interrupt generation condition.

typedef enum *rgpio_interrupt_config* `rgpio_interrupt_config_t`

Configures the interrupt generation condition.

typedef struct *rgpio_pin_config* `rgpio_pin_config_t`

The RGPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the `outputConfig` unused. Note that in some use cases, the corresponding port property should be configured in advance with the `PORT_SetPinConfig()`.

struct _rgpio_pin_config

#include <fsl_rgpio.h> The RGPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

Public Members

rgpio_pin_direction_t pinDirection

RGPIO direction, input or output

uint8_t outputLogic

Set a default output logic, which has no use in input

2.63 RGPIO Driver

void RGPIO_PinInit(RGPIO_Type *base, uint32_t pin, const rgpio_pin_config_t *config)

Initializes a RGPIO pin used by the board.

To initialize the RGPIO, define a pin configuration, as either input or output, in the user file. Then, call the RGPIO_PinInit() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
rgpio_pin_config_t config =
{
    kRGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
rgpio_pin_config_t config =
{
    kRGPIO_DigitalOutput,
    0,
}
```

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- pin – RGPIO port pin number
- config – RGPIO pin configuration pointer

uint32_t RGPIO_GetInstance(RGPIO_Type *base)

Gets the RGPIO instance according to the RGPIO base.

Parameters

- base – RGPIO peripheral base pointer(PTA, PTB, PTC, etc.)

Return values

RGPIO – instance

static inline void RGPIO_PinWrite(RGPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the multiple RGPIO pins to the logic 1 or 0.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- pin – RGPIO pin number
- output – RGPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void RGPIO_WritePinOutput(RGPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the multiple RGPIO pins to the logic 1 or 0.

Deprecated:

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

static inline void RGPIO_PortSet(RGPIO_Type *base, uint32_t mask)

Sets the output level of the multiple RGPIO pins to the logic 1.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number macro

static inline void RGPIO_SetPinsOutput(RGPIO_Type *base, uint32_t mask)

Sets the output level of the multiple RGPIO pins to the logic 1.

Deprecated:

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

static inline void RGPIO_PortClear(RGPIO_Type *base, uint32_t mask)

Sets the output level of the multiple RGPIO pins to the logic 0.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number macro

static inline void RGPIO_ClearPinsOutput(RGPIO_Type *base, uint32_t mask)

Sets the output level of the multiple RGPIO pins to the logic 0.

Deprecated:

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

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number macro

static inline void RGPIO_PortToggle(RGPIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple RGPIO pins.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)

- mask – RGPIO pin number macro

static inline void RGPIO_TogglePinsOutput(RGPIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple RGPIO pins.

Deprecated:

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

static inline uint32_t RGPIO_PinRead(RGPIO_Type *base, uint32_t pin)

Reads the current input value of the RGPIO port.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- pin – RGPIO pin number

Return values

RGPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline uint32_t RGPIO_ReadPinInput(RGPIO_Type *base, uint32_t pin)

Reads the current input value of the RGPIO port.

Deprecated:

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

static inline void RGPIO_EnablePortInput(RGPIO_Type *base, uint32_t mask, bool enable)

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number mask
- enable – RGPIO digital input enable/disable flag.

void RGPIO_CheckAttributeBytes(RGPIO_Type *base, *rgpio_checker_attribute_t* attribute)

The RGPIO module supports a device-specific number of data ports, organized as 32-bit words. Each 32-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the RGPIO programming model. The attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number macro

static inline void RGPIO_SetPinInterruptConfig(RGPIO_Type *base, uint32_t pin,
rgpio_interrupt_sel_t sel,
rgpio_interrupt_config_t config)

Configures the gpio pin interrupt/DMA request.

Parameters

- base – RGPIO peripheral base pointer.
- pin – RGPIO pin number.

- sel – RGPIO pin interrupt selection(0-3).
- config – RGPIO pin interrupt configuration.

```
static inline void __SetMultipleInterruptPinsConfig(RGPIO_Type *base, uint32_t mask,  
                                                    rgpio_interrupt_sel_t sel,  
                                                    rgpio_interrupt_config_t config)
```

Sets the gpio interrupt configuration in ICR register for multiple pins.

Parameters

- base – RGPIO peripheral base pointer (RGPIOA, RGPIOB, RGPIOC, and so on.)
- mask – RGPIO pin number macro.
- sel – RGPIO pin interrupt selection(0-3).
- config – RGPIO pin interrupt configuration.

```
static inline uint32_t RGPIO_GetPinsInterruptFlags(RGPIO_Type *base, rgpio_interrupt_sel_t sel)
```

Reads the whole gpio status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

- base – RGPIO peripheral base pointer.
- sel – RGPIO pin interrupt selection(0-3).

Returns

Current gpio interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

```
static inline void RGPIO_ClearPinsInterruptFlags(RGPIO_Type *base, rgpio_interrupt_sel_t sel,  
                                                  uint32_t mask)
```

Clears the multiple pin interrupt status flag.

Parameters

- base – RGPIO peripheral base pointer.
- sel – RGPIO pin interrupt selection(0-3).
- mask – RGPIO pin number macro.

2.64 RTD_CMC: Real Time Domain Core Mode Controller Driver

```
void RTDCMC_SetClockMode(CMC_Type *base, cmc_clock_mode_t mode)
```

Sets clock mode.

This function config the amount of clock gating when the core asserts Sleeping due to WFI, WFE or SLEEPONEXIT.

Parameters

- base – CMC peripheral base address.
- mode – System clock mode.

```
static inline void RTDCMC_LockClockModeSetting(CMC_Type *base)
```

Locks the clock mode setting.

After invoking this function, any clock mode setting will be blocked.

Parameters

- base – CMC peripheral base address.

```
static inline cmc_core_clock_gate_status_t RTDCMC_GetCoreClockGatedStatus(CMC_Type  
*base)
```

Gets the core clock gated status.

This function get the status to indicate whether the core clock is gated. The core clock gated status can be cleared by software.

Parameters

- base – CMC peripheral base address.

Returns

The status to indicate whether the core clock is gated.

```
static inline void RTDCMC_ClearCoreClockGatedStatus(CMC_Type *base)
```

Clears the core clock gated status.

This function clear clock status flag by software.

Parameters

- base – CMC peripheral base address.

```
static inline uint8_t RTDCMC_GetWakeupSource(CMC_Type *base)
```

Gets the Wakeup Source.

This function gets the Wakeup sources from the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns

The Wakeup sources from the previous low power mode entry. See `_rtd_cmc_wakeup_sources` for details.

```
static inline cmc_clock_mode_t RTDCMC_GetClockMode(CMC_Type *base)
```

Gets the Clock mode.

This function gets the clock mode of the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns

The Low Power status.

```
static inline uint32_t RTDCMC_GetSystemResetStatus(CMC_Type *base)
```

Gets the System reset status.

This function returns the system reset status. Those status updates on every MAIN Warm Reset to indicate the type/source of the most recent reset.

Parameters

- base – CMC peripheral base address.

Returns

The most recent system reset status. See `_rtd_cmc_system_reset_sources` for details.

```
static inline uint32_t RTDCMC_GetStickySystemResetStatus(CMC_Type *base)
```

Gets the sticky system reset status since the last WAKE Cold Reset.

This function gets all source of system reset that have generated a system reset since the last WAKE Cold Reset, and that have not been cleared by software.

Parameters

- base – CMC peripheral base address.

Returns

System reset status that have not been cleared by software. See `_rtd_cmc_system_reset_sources` for details.

```
static inline void RTDCMC_ClearStickySystemResetStatus(CMC_Type *base, uint32_t mask)
```

Clears the sticky system reset status flags.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the sticky system reset status to be cleared.

```
void RTDCMC_SetPowerModeProtection(CMC_Type *base, uint8_t allowedModes)
```

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes. This should be done before set the lowPower mode for each power doamin.

The allowed lowpower modes are passed as bit map. For example, to allow Sleep and DeepSleep, use `CMC_SetPowerModeProtection(CMC_base, kRTDCMC_AllowSleepMode | kRTDCMC_AllowDeepSleepMode)`. To allow all low power modes, use `CMC_SetPowerModeProtection(CMC_base, kRTDCMC_AllowAllLowPowerModes)`.

Parameters

- base – CMC peripheral base address.
- allowedModes – Bitmaps of the allowed power modes. See `_rtd_cmc_power_mode_protection` for details.

```
static inline void RTDCMC_LockPowerModeProtectionSetting(CMC_Type *base)
```

Locks the power mode protection.

This function locks the power mode protection. After invoking this function, any power mode protection setting will be ignored.

Parameters

- base – CMC peripheral base address.

```
void RTDCMC_EnterLowPowerMode(CMC_Type *base, cmc_low_power_mode_t lowPowerMode)
```

Enter into the selected low power mode, please make sure the selected power mode has been enabled in the protection level.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See `cmc_low_power_mode_t` for details.

```
static inline void RTDCMC_EnableSystemResetInterrupt(CMC_Type *base, uint32_t mask)
```

Enable system reset interrupts.

This function enables the system reset interrupts. The assertion of non-fatal warm reset can be delayed for 258 cycles of the 32K_CLK clock while an enabled interrupt is generated. Then Software can perform a graceful shutdown or abort the non-fatal warm reset

provided the pending reset source is cleared by resetting the reset source and then clearing the pending flag.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See `_rtd_cmc_system_reset_interrupt_enable` for details.

```
static inline void RTDCMC_DisableSystemResetInterrupt(CMC_Type *base, uint32_t mask)
```

Disable system reset interrupts.

This function disables the system reset interrupts.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See `_rtd_cmc_system_reset_interrupt_enable` for details.

```
static inline uint32_t RTDCMC_GetSystemResetInterruptFlags(CMC_Type *base)
```

Gets System Reset interrupt flags.

This function returns the System reset interrupt flags.

Parameters

- base – CMC peripheral base address.

Returns

System reset interrupt flags. See `_rtd_cmc_system_reset_interrupt_flag` for details.

```
static inline void RTDCMC_ClearSystemResetInterruptFlags(CMC_Type *base, uint32_t mask)
```

Clears System Reset interrupt flags.

This function clears system reset interrupt flags. The pending reset source can be cleared by resetting the source of the reset and then clearing the pending flags.

Parameters

- base – CMC peripheral base address.
- mask – System Reset interrupt flags. See `_rtd_cmc_system_reset_interrupt_flag` for details.

```
static inline void RTDCMC_EnablePowerSwitchDomainOutOfResetInterrupt(CMC_Type *base,  
uint32_t mask)
```

Enable power switch domain out of reset interrupts.

Parameters

- base – CMC peripheral base address.
- mask – Power switch domain out of reset interrupt mask, should be the OR'ed value of `_rtd_cmc_power_switch_domain_out_of_reset_interrupt_enable`.

```
static inline void RTDCMC_DisablePowerSwitchDomainOutOfResetInterrupt(CMC_Type *base,  
uint32_t mask)
```

Disable power switch domain out of reset interrupts.

Parameters

- base – CMC peripheral base address.

- mask – Power switch domain out of reset interrupt mask, should be the OR'ed value of `_rtd_cmc_power_switch_domain_out_of_reset_interrupt_enable`.

```
static inline uint32_t RTDCMC_GetPowerSwitchDomainOutOfResetInterruptFlags(CMC_Type *base)
```

Get power switch domain out of reset interrupt flags.

Parameters

- base – CMC peripheral base address.

Returns

Power switch domain out of reset interrupt flags, shall be the OR'ed value of `_rtd_cmc_power_switch_domain_out_of_reset_interrupt_flag`.

```
static inline void RTDCMC_ClearPowerSwitchDomainOutOfResetInterruptFlags(CMC_Type *base,
                                                                           uint32_t mask)
```

Clear power switch domain out of reset interrupt flags.

Parameters

- base – CMC peripheral base address.
- mask – The mask of power switch domain out of reset interrupt flags to clear.

```
static inline bool RTDCMC_CheckRealTimeDomainPowerSwitchDomainStatus(CMC_Type *base,
                                                                       rtd_cmc_power_switch_domain_name_t
                                                                       domainName)
```

Check the status of selected power switch domain.

Parameters

- base – CMC peripheral base address.
- domainName – The selected domain name, please refer to `rtd_cmc_power_switch_domain_name_t`.

Return values

- True – The power switch domain is open.
- False – The power switch domain is closed.

```
static inline uint8_t RTDCMC_GetBootConfigPinLogic(CMC_Type *base)
```

Gets the logic state of the `BOOT_CONFIGn` pin.

This function returns the logic state of the `BOOT_CONFIGn` pin on the last negation of `RESET_b` pin.

Parameters

- base – CMC peripheral base address.

Returns

The logic state of the `BOOT_CONFIGn` pin on the last negation of `RESET_b` pin.

```
static inline void RTDCMC_ClearBootConfig(CMC_Type *base)
```

Clears `BOOT_CONFIG` pin state.

Parameters

- base – CMC peripheral base address.

```
static inline void RTDCMC_ForceBootConfiguration(CMC_Type *base, bool assert, uint32_t
                                              bitsMask)
```

Set the logic state of the BOOT_CONFIGn pin.

This function force the logic state of the Boot_Config pin to assert on next system reset.

Parameters

- base – CMC peripheral base address.
- assert – Assert the corresponding pin or not. true - Assert corresponding pin on next system reset. false - No effect.
- bitsMask – The mask of the corresponding bits in the Mode register to assert or not on the next system reset.

```
static inline void RTDCMC_EnableNonMaskablePinInterrupt(CMC_Type *base, bool enable)
Enable/Disable Non maskable Pin interrupt.
```

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Non maskable pin interrupt. true - enable Non-maskable pin interrupt. false - disable Non-maskable pin interrupt.

```
static inline void RTDCMC_EnableDebugOperation(CMC_Type *base, bool enable)
Enables/Disables debug Operation when the core sleep.
```

This function configs what happens to debug when core sleeps.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Debug when Core is sleeping. true - Debug remains enabled when the core is sleeping. false - Debug is disabled when the core is sleeping.

```
FSL_RTD_CMC_DRIVER_VERSION
RTD_CMC driver version 2.0.2.
```

```
enum _rtd_cmc_power_mode_protection
RTD_CMC power mode Protection enumeration.
```

Values:

```
enumerator kRTDCMC_AllowSleepMode
Allow Sleep mode.
```

```
enumerator kRTDCMC_AllowDeepSleepMode
Allow Deep Sleep mode.
```

```
enumerator kRTDCMC_AllowPowerDownMode
Allow Power Down mode.
```

```
enumerator kRTDCMC_AllowDeepPowerDownMode
Allow Deep Power Down mode.
```

```
enumerator kRTDCMC_AllowHoldMode
Allow Hold mode.
```

```
enumerator kRTDCMC_AllowAllLowPowerModes
Allow all low power modes.
```

enum _rtd_cmc_wakeup_sources

Wake up sources from the previous low power mode entry.

Values:

enumerator kRTDCMC_WakeupFromResetInterruptOrPowerDown

Wakeup source is reset interrupt, or wake up from [Deep] Power Down.

enumerator kRTDCMC_WakeupFromDebugReuquest

Wakeup source is debug request.

enumerator kRTDCMC_WakeupFromInterrupt

Wakeup source is interrupt.

enumerator kRTDCMC_WakeupFromDMAWakeup

Wakeup source is DMA Wakeup.

enumerator kRTDCMC_WakeupFromWUURequest

Wakeup source is WUU request.

enum _rtd_cmc_system_reset_interrupt_enable

System Reset Interrupt enable enumeration.

Values:

enumerator kRTDCMC_InSystemProgrammingAccessPortResetInterruptEnable

ISP_AP Reset interrupt enable.

enumerator kRTDCMC_LowPowerAcknowledgeTimeoutResetInterruptEnable

Low Power Acknowledge Timeout Reset interrupt enable.

enumerator kRTDCMC_Watchdog0ResetInterruptEnable

Watchdog 0 Reset interrupt enable.

enumerator kRTDCMC_SoftwareResetInterruptEnable

Software Reset interrupt enable.

enumerator kRTDCMC_LockupResetInterruptEnable

Lockup Reset interrupt enable.

enumerator kRTDCMC_AppDomainMuSystemResetInterruptEnable

enumerator kRTDCMC_Watchdog1ResetInterruptEnable

Watchdog 1 Reset interrupt enable.

enumerator kRTDCMC_Watchdog5ResetInterruptEnable

Watchdog 5 Reset interrupt enable.

enumerator kRTDCMC_Watchdog2ResetInterruptEnable

Watchdog 2 Reset interrupt enable.

enum _rtd_cmc_power_switch_domain_out_of_reset_interrupt_enable

Power switch domain out of reset interrupt enable enumeration.

Values:

enumerator kRTDCMC_FusionPSDomainOutOfResetInterruptEnable

Fusion power switch domain out of reset interrupt enable.

enumerator kRTDCMC_FusionAlwaysOnPSDomainOutOfResetInterruptEnable

Fusion always on power switch domain out of reset interrupt enable.

enum _rtd_cmc_system_reset_interrupt_flag

CMC System Reset Interrupt Status flag.

Values:

enumerator kRTDCMC_InSystemProgrammingAccessPortResetDAPResetInterruptFlag
DAP Reset interrupt flag.

enumerator kRTDCMC_LowPowerAcknowledgeTimeoutResetFlag
Low Power Acknowledge Timeout Reset interrupt flag.

enumerator kRTDCMC_Watchdog0ResetInterruptFlag
Watchdog 0 Reset interrupt flag.

enumerator kRTDCMC_SoftwareResetInterruptFlag
Software Reset interrupt flag.

enumerator kRTDCMC_LockupResetInterruptFlag
Lock up Reset interrupt flag.

enumerator kRTDCMC_AppDomainMuSystemResetInterruptFlag
Application Domain MU System Reset interrupt flag.

enumerator kRTDCMC_WatchdogSResetInterruptFlag
Watchdog S Reset interrupt flag.

enumerator kRTDCMC_WatchdogHifi4ResetInterruptFlag
Watchdog Hifi4 Reset interrupt flag.

enumerator kRTDCMC_WatchdogFusionResetInterruptFlag
Watchdog Fusion Reset interrupt flag.

enum _rtd_cmc_power_switch_domain_out_of_reset_interrupt_flag
Power switch domain out of reset interrupt flags enumeration.

Values:

enumerator kRTDCMC_FusionPSDomainOutOfResetInterruptFlag
Fusion power switch domain out of reset interrupt flag.

enumerator kRTDCMC_FusionAlwaysOnPSDomainOutOfResetInterruptFlag
Fusion always on power switch domain out of reset interrupt flag.

enum _rtd_cmc_system_reset_sources
System reset sources enumeration.

Values:

enumerator kRTDCMC_WakeUpReset
The reset caused by a wakeup from Power Down or Deep Power Down mode.

enumerator kRTDCMC_PORReset
The reset caused by power on reset detection logic.

enumerator kRTDCMC_HLVDRReset
The reset caused by a High or Low Voltage Detect.

enumerator kRTDCMC_WarmReset
The last reset source is a warm reset source.

enumerator kRTDCMC_FatalReset
The last reset source is a fatal reset source.

enumerator kRTDCMC_PinReset

The reset caused by the RESET_b pin.

enumerator kRTDCMC_ISP_APRReset

The reset caused by a reset request from in-system programming access port.

enumerator kRTDCMC_ResetTimeout

The reset caused by a timeout or other error condition in the system reset generation.

enumerator kRTDCMC_LowPowerAcknowledgeTimeoutReset

The reset caused by a timeout in low power mode entry logic.

enumerator kRTDCMC_RtdCgcLocReset

The reset caused by real time domain clock generation and control loss-of-clock reset.

enumerator kRTDCMC_Watchdog0Reset

The reset caused by a WatchDog 0 timeout.

enumerator kRTDCMC_SoftwareReset

The reset caused by a software reset request.

enumerator kRTDCMC_LockUpReset

The reset caused by the ARM core indication of a LOCKUP event.

enumerator kRTDCMC_AdMuReset

The reset caused by application domain MU system.

enumerator kRTDCMC_RtdCgcLosReset

The reset caused by RTD clock generation and control loss of sync.

enumerator kRTDCMC_LpavCgcLosReset

The reset caused by LPAV clock generation and control loss of sync.

enumerator kRTDCMC_uPowerReset

The reset caused by uPOWER WDOG System.

enumerator kRTDCMC_VbatReset

The reset caused by VBAT system reset.

enumerator kRTDCMC_WatchdogSReset

The reset caused by a WatchDog_S RTD timeout.

enumerator kRTDCMC_WatchdogHifi4Reset

The reset caused from the WDOG_HIFI4 timeout.

enumerator kRTDCMC_WatchdogFusionReset

The reset caused from the WDOG_FUSION timeout.

enumerator kRTDCMC_JTAGSystemReset

The reset caused by a JTAG system reset request.

enumerator kRTDCMC_SentinelLifeCycleReset

The reset caused by sentinel detecting an unexpected lifecycle coding.

enumerator kRTDCMC_SentinelReset

The reset caused by sentinel reset request.

enumerator kRTDCMC_SentinelSystemFailReset

The reset caused by a sentinel system fail condition.

enum _rtd_cmc_core_clock_gate_status

Indicate the core clock was gated.

Values:

enumerator kRTDCMC_CoreClockNotGated

Core clock not gated.

enumerator kRTDCMC_CoreClockGated

Core clock was gated due to low power mode entry.

enum _rtd_cmc_clock_mode

CMC clock mode enumeration.

Values:

enumerator kRTDCMC_GateNoneClock

No clock gating.

enumerator kRTDCMC_GateCoreClock

Gate Core clock.

enumerator kRTDCMC_GateCorePlatformClock

Gate Core clock and platform clock.

enumerator kRTDCMC_GateAllSystemClocks

Gate all System clocks, without getting core entering into low power mode.

enum _rtd_cmc_low_power_mode

CMC power mode enumeration.

Values:

enumerator kRTDCMC_ActiveMode

Select Active mode.

enumerator kRTDCMC_SleepMode

Select Sleep mode when a core executes WFI or WFE instruction.

enumerator kRTDCMC_DeepSleepMode

Select Deep Sleep mode when a core executes WFI or WFE instruction.

enumerator kRTDCMC_PowerDownMode

Select Power Down mode when a core executes WFI or WFE instruction.

enumerator kRTDCMC_DeepPowerDown

Select Deep Power Down mode when a core executes WFI or WFE instruction.

enumerator kRTDCMC_HoldMode

Select Hold mode

enum _rtd_cmc_power_switch_domain_name

Values:

enumerator kRTDCMC_FusionAlwaysOnPowerSwitchDomain

Fusion always on power switch domain.

enumerator kRTDCMC_FusionPowerSwitchDomain

Fusion power switch domain.

enumerator kRTDCMC_RealtimePowerDomain

Realtime power switch domain

typedef enum _rtd_cmc_core_clock_gate_status cmc_core_clock_gate_status_t

Indicate the core clock was gated.

typedef enum _rtd_cmc_clock_mode cmc_clock_mode_t

CMC clock mode enumeration.

```
typedef enum _rtd_cmc_low_power_mode cmc_low_power_mode_t
```

CMC power mode enumeration.

```
typedef enum _rtd_cmc_power_switch_domain_name rtd_cmc_power_switch_domain_name_t
```

2.65 SAI: Serial Audio Interface

2.66 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 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 Rx 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 2.4.8

_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_blk_source
Bit clock source.

Values:

enumerator kSAI_BlkSourceBusclk
Bit clock using bus clock

enumerator kSAI_BlkSourceMclkOption1
Bit clock MCLK option 1

enumerator kSAI_BlkSourceMclkOption2
Bit clock MCLK option2

enumerator kSAI_BlkSourceMclkOption3
Bit clock MCLK option3

enumerator kSAI_BlkSourceMclkDiv
Bit clock using master clock divider

enumerator kSAI_BlkSourceOtherSai0
Bit clock from other SAI device

enumerator kSAI_BlkSourceOtherSai1
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.

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 waring 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

2.67 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,

- for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
- 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

2.68 SEMA42: Hardware Semaphores Driver

`FSL_SEMA42_DRIVER_VERSION`
SEMA42 driver version.

SEMA42 status return codes.

Values:

enumerator `kStatus_SEMA42_Busy`
SEMA42 gate has been locked by other processor.

enumerator `kStatus_SEMA42_Reseting`
SEMA42 gate reseting is ongoing.

enum `_sema42_gate_status`
SEMA42 gate lock status.

Values:

enumerator `kSEMA42_Unlocked`
The gate is unlocked.

enumerator kSEMA42_LockedByProc0

The gate is locked by processor 0.

enumerator kSEMA42_LockedByProc1

The gate is locked by processor 1.

enumerator kSEMA42_LockedByProc2

The gate is locked by processor 2.

enumerator kSEMA42_LockedByProc3

The gate is locked by processor 3.

enumerator kSEMA42_LockedByProc4

The gate is locked by processor 4.

enumerator kSEMA42_LockedByProc5

The gate is locked by processor 5.

enumerator kSEMA42_LockedByProc6

The gate is locked by processor 6.

enumerator kSEMA42_LockedByProc7

The gate is locked by processor 7.

enumerator kSEMA42_LockedByProc8

The gate is locked by processor 8.

enumerator kSEMA42_LockedByProc9

The gate is locked by processor 9.

enumerator kSEMA42_LockedByProc10

The gate is locked by processor 10.

enumerator kSEMA42_LockedByProc11

The gate is locked by processor 11.

enumerator kSEMA42_LockedByProc12

The gate is locked by processor 12.

enumerator kSEMA42_LockedByProc13

The gate is locked by processor 13.

enumerator kSEMA42_LockedByProc14

The gate is locked by processor 14.

typedef enum *sema42_gate_status* sema42_gate_status_t

SEMA42 gate lock status.

void SEMA42_Init(SEMA42_Type *base)

Initializes the SEMA42 module.

This function initializes the SEMA42 module. It only enables the clock but does not reset the gates because the module might be used by other processors at the same time. To reset the gates, call either SEMA42_ResetGate or SEMA42_ResetAllGates function.

Parameters

- base – SEMA42 peripheral base address.

void SEMA42_Deinit(SEMA42_Type *base)

De-initializes the SEMA42 module.

This function de-initializes the SEMA42 module. It only disables the clock.

Parameters

- base – SEMA42 peripheral base address.

status_t SEMA42_TryLock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Tries to lock the SEMA42 gate.

This function tries to lock the specific SEMA42 gate. If the gate has been locked by another processor, this function returns an error code.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – Lock the sema42 gate successfully.
- kStatus_SEMA42_Busy – Sema42 gate has been locked by another processor.

status_t SEMA42_Lock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Locks the SEMA42 gate.

This function locks the specific SEMA42 gate. If the gate has been locked by other processors, this function waits until it is unlocked and then lock it.

If SEMA42_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – The gate was successfully locked.
- kStatus_Timeout – Timeout occurred while waiting for the gate to be unlocked.

Returns

status_t

static inline void SEMA42_Unlock(SEMA42_Type *base, uint8_t gateNum)

Unlocks the SEMA42 gate.

This function unlocks the specific SEMA42 gate. It only writes unlock value to the SEMA42 gate register. However, it does not check whether the SEMA42 gate is locked by the current processor or not. As a result, if the SEMA42 gate is not locked by the current processor, this function has no effect.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to unlock.

static inline *sema42_gate_status_t* SEMA42_GetGateStatus(SEMA42_Type *base, uint8_t gateNum)

Gets the status of the SEMA42 gate.

This function checks the lock status of a specific SEMA42 gate.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number.

Returns

status Current status.

status_t SEMA42_ResetGate(SEMA42_Type *base, uint8_t gateNum)

Resets the SEMA42 gate to an unlocked status.

This function resets a SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number.

Return values

- kStatus_Success – SEMA42 gate is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

static inline *status_t* SEMA42_ResetAllGates(SEMA42_Type *base)

Resets all SEMA42 gates to an unlocked status.

This function resets all SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.

Return values

- kStatus_Success – SEMA42 is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

SEMA42_GATE_NUM_RESET_ALL

The number to reset all SEMA42 gates.

SEMA42_GATEn(base, n)

SEMA42 gate n register address.

The SEMA42 gates are sorted in the order 3, 2, 1, 0, 7, 6, 5, 4, ... not in the order 0, 1, 2, 3, 4, 5, 6, 7, ... The macro SEMA42_GATEn gets the SEMA42 gate based on the gate index.

The input gate index is XOR'ed with 3U: $0 \wedge 3 = 3$ $1 \wedge 3 = 2$ $2 \wedge 3 = 1$ $3 \wedge 3 = 0$ $4 \wedge 3 = 7$ $5 \wedge 3 = 6$ $6 \wedge 3 = 5$ $7 \wedge 3 = 4$...

SEMA42_BUSY_POLL_COUNT

Maximum polling iterations for SEMA42 waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the SEMA42 driver code before timing out and returning an error.

It applies to all waiting loops in SEMA42 driver, such as waiting for a gate to be unlocked, waiting for a reset to complete, or waiting for a resource to become available.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if hardware doesn't respond or if a resource is never released.

2.69 Sentinel

enum rdc_type

Values:

enumerator TRDC_TYPE

enumerator XRDC_TYPE

void SENTINEL_Init(void)

Initialize MU interface for Sentinel access.

void SENTINEL_Deinit(void)

Deinitialize MU interface for Sentinel access.

void SENTINEL_SendMessage(uint32_t *param, uint32_t paramCount)

Send message to Sentinel.

Parameters

- param – Command parameters pointer. Each parameter is a 32bit word.
- paramCount – Command parameter count.

void SENTINEL_ReceiveMessage(uint32_t *pParam, uint32_t *pParamCount)

Receive message from Sentinel.

Parameters

- pParam – Pointer to save command parameters. Each parameter is a 32bit word.
- pParamCount – Inout pointer to command parameter count.

uint32_t SENTINEL_Command(uint32_t *cmdParam, uint32_t cmdParamCount, uint32_t *pRespParam, uint32_t *pRespParamCount)

Send command to Sentinel and receive response.

Parameters

- cmdParam – Command parameters pointer. Each parameter is a 32bit word.
- cmdParamCount – Command parameter count.
- pRespParam – Pointer to save response parameters. Each parameter is a 32bit word.
- pRespParamCount – Inout pointer to response parameter count.

Returns

0 for success, otherwise return error code.

uint32_t SENTINEL_Ping(void)

Ping Sentinel to see if it is alive.

Returns

0 for success, otherwise return error code.

uint32_t SENTINEL_GetFirmwareVersion(uint32_t *pVersion, uint32_t *pCommitSha)

Get Sentinel firmware version.

Parameters

- pVersion – Pointer to save firmware version.
- pCommitSha – Pointer to save first 4 bytes of the git commit ID.

Returns

0 for success, otherwise return error code.

uint32_t SENTINEL_GetFirmwareStatus(uint8_t *pStatus)

Get Sentinel firmware status.

Parameters

- pStatus – Pointer to save firmware status. 0: No firmware in place. 1: Firmware authenticated and operational.

Returns

0 for success, otherwise return error code.

uint32_t SENTINEL_SetPowerDown(uint32_t resumeAddr, uint32_t size)

Request Sentinel to enter power down.

Parameters

- resumeAddr – Base address in memory for resume code
- size – Length of resume code in memory

Returns

0 for success, otherwise return error code.

uint32_t SENTINEL_SessionOpen(uint32_t *pSessionHandle, uint8_t muId, uint8_t interruptId, uint8_t tz, uint8_t did, uint8_t priority, uint8_t operatingMode)

uint32_t SENTINEL_SessionClose(uint32_t *pSessionHandle)

uint32_t SENTINEL_RngOpen(uint32_t *pRngHandle, uint32_t sessionHandle, uint8_t flags)

uint32_t SENTINEL_RngClose(uint32_t *pRngHandle)

uint32_t SENTINEL_RngGetRandom(uint32_t *pRngHandle, uint32_t outAddr, uint32_t outSize)

status_t SENTINEL_RNG_GetRandomData(uint32_t output, uint32_t len)

uint32_t SENTINEL_ReleaseRDC(enum *rdc_type* type)

uint32_t SENTINEL_GetSocInfo(uint16_t *pSoc_id, uint16_t *pSoc_rev)

Get Sentinel cpu version.

Parameters

- pSoc_id – Pointer to soc id.
- pSoc_rev – Pointer to soc revision number.

Returns

0 for success, otherwise return error code.

FSL_SENTINEL_DRIVER_VERSION

sentinel driver version 2.0.3.

SENTINEL_BASELINE_API_CMD_PING

sentinel command definition.

SENTINEL_BASELINE_API_CMD_DUMP_DEBUG_BUFFER

SENTINEL_BASELINE_API_CMD_OEM_CNTN_AUTH

SENTINEL_BASELINE_API_CMD_VERIFY_IMAGE

SENTINEL_BASELINE_API_CMD_RELEASE_CONTAINER

SENTINEL_BASELINE_API_CMD_GET_FW_VERSION

SENTINEL_BASELINE_API_CMD_RELEASE_RDC

SENTINEL_BASELINE_API_CMD_GET_FW_STATUS

SENTINEL_BASELINE_API_CMD_POWER_DOWN

SENTINEL_BASELINE_API_CMD_GET_INFO

SENTINEL_BRIDGE_API_CMD_SESSION_OPEN

SENTINEL_BRIDGE_API_CMD_SESSION_CLOSE

SENTINEL_BRIDGE_API_CMD_RNG_OPEN

SENTINEL_BRIDGE_API_CMD_RNG_CLOSE

SENTINEL_BRIDGE_API_CMD_RNG_GET_RANDOM

SENTINEL_MSG_HDR_CMD_TAG

SENTINEL_MSG_HDR_CMD_RSP_TAG

SENTINEL_BASELINE_API_VER

SENTINEL_BRIDGE_API_VER

SENTINEL_RR_CNT

SENTINEL_TR_CNT

RTC_MU_ID

AP_MU_ID

CM33_DOMAIN_ID

BASELINE_SUCCESS_IND

BASELINE_FAILURE_IND

BRIDGE_SUCCESS_STATUS

BRIDGE_FAILURE_STATUS

SENTINEL_ACCESS_ADDR_ALIGN_SIZE

RTD_CORE_ID

APD_CORE_ID

TRDC_ID

XRDC_ID

uint8_t ver

uint8_t size

uint8_t cmd

uint8_t tag

uint8_t status

uint8_t indication

uint16_t abort__code

uint8_t status

uint8_t rating

uint16_t rating__ext

uint32_t code

struct *baseline_api_rsp_code* baseline__rsp__code

struct *bridge_api_rsp_code* bridge__rsp__code

struct *msg_hdr* hdr

union *response_code* rsp__code

struct *msg_hdr* hdr

struct *common_rsp* rsp

uint32_t fw__version

uint32_t commit__id

struct *msg_hdr* hdr

struct *common_rsp* rsp

uint8_t status

uint8_t rsv[3]

struct *msg_hdr* hdr

uint8_t mu__id

uint8_t interrupt__num

uint8_t tz

uint8_t did

uint8_t priority

uint8_t op__mode

uint16_t reserved

struct *common_rsp* rsp

uint32_t session__handle

struct *msg_hdr* hdr

uint32_t size

uint32_t resume__addr

struct *common_rsp* rsp

struct *msg_hdr* hdr

```
uint32_t session_handle
struct common_rsp rsp
struct msg_hdr hdr
uint32_t session_handle
uint32_t input_address_ext
uint32_t output_address_ext
uint8_t flags
uint8_t rsv[3]
uint32_t crc
struct common_rsp rsp
uint32_t rng_handle
struct msg_hdr hdr
uint32_t rng_handle
struct common_rsp rsp
struct msg_hdr hdr
uint32_t rng_handle
uint32_t out_addr
uint32_t out_size
struct common_rsp rsp
struct msg_hdr hdr
struct common_rsp rsp
uint32_t debug_words[20]
uint32_t crc
struct msg_hdr hdr
uint8_t core_id
uint8_t rdc_id
uint8_t rsv[2]
struct common_rsp rsp
struct msg_hdr hdr
uint32_t rsp_msb
uint32_t rsp_lsb
uint16_t buf_size
uint16_t rsv
```

```
uint8_t cmd
uint8_t ver
uint16_t len
uint16_t soc_id
uint16_t soc_rev
uint16_t lc
uint8_t sssm
uint8_t rsv
uint32_t uid[4]
uint32_t sha256_rom_patch[8]
uint32_t sha_fw[8]
uint32_t oem_srkh[16]
uint8_t trng
uint8_t csal
uint8_t imem
uint8_t rsv2
struct msg_hdr
    #include <fsl_sentinel.h>
struct baseline_api_rsp_code
    #include <fsl_sentinel.h>
struct bridge_api_rsp_code
    #include <fsl_sentinel.h>
union response_code
    #include <fsl_sentinel.h>
struct common_rsp
    #include <fsl_sentinel.h>
struct get_fw_version_msg_cmd
    #include <fsl_sentinel.h>
struct get_fw_version_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct get_fw_status_msg_cmd
    #include <fsl_sentinel.h>
struct get_fw_status_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct open_session_msg_cmd
    #include <fsl_sentinel.h>
struct open_session_msg_cmd_rsp
    #include <fsl_sentinel.h>
```

```
struct power_down_cmd
    #include <fsl_sentinel.h>
struct power_down_cmd_rsp
    #include <fsl_sentinel.h>
struct close_session_msg_cmd
    #include <fsl_sentinel.h>
struct close_session_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct rng_open_msg_cmd
    #include <fsl_sentinel.h>
struct rng_open_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct rng_close_msg_cmd
    #include <fsl_sentinel.h>
struct rng_close_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct rng_get_rnd_msg_cmd
    #include <fsl_sentinel.h>
struct rng_get_rnd_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct dump_debug_buffer_msg_cmd
    #include <fsl_sentinel.h>
struct dump_debug_buffer_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct release_rdc_msg_cmd
    #include <fsl_sentinel.h>
struct release_rdc_msg_cmd_rsp
    #include <fsl_sentinel.h>
struct get_info_msg_cmd
    #include <fsl_sentinel.h>
struct sentinel_get_info_data
    #include <fsl_sentinel.h>
```

2.70 SFA: Signal Frequency Analyser

`void SFA_GetDefaultConfig(sfa_config_t *config)`
Fill the SFA configuration structure with default settings.
The default values are:

```
config->mode = kSFA_FrequencyMeasurement0;
config->cutSelect = kSFA_CUTSelect0;
config->refSelect = kSFA_REFSelect0;
config->prediv = 0U;
config->trigStart = kSFA_TriggerStartSelect0;
```

(continues on next page)

(continued from previous page)

```

config->startPolarity = kSFA_TriggerStartPolarityRiseEdge;
config->trigEnd = kSFA_TriggerEndSelect0;
config->endPolarity = kSFA_TriggerEndPolarityRiseEdge;
config->enableTrigMeasurement = false;
config->enableCUTPin = false;
config->cutTarget = 0xffffU;
config->refTarget = 0xffffffffU;

```

Parameters

- config – Pointer to the user configuration structure.

void SFA_Init(SFA_Type *base)

Initialize SFA.

Parameters

- base – SFA peripheral base address.

status_t SFA_Deinit(SFA_Type *base)

Clear counter, disable SFA and gate the SFA clock.

Parameters

- base – SFA peripheral base address.

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

static inline void SFA_EnableCUTPin(SFA_Type *base, bool enable)

Control the connection of the clock under test to an external pin.

Parameters

- base – SFA peripheral base address.
- enable – true: connect the clock under test and external pin. false: Disconnect the clock under test and external pin.

static inline uint32_t SFA_GetStatusFlags(SFA_Type *base)

Get SFA status flags.

Parameters

- base – SFA peripheral base address.

void SFA_ClearStatusFlag(SFA_Type *base, uint32_t mask)

Clear the SFA status flags.

Note: To clear kSFA_RefStoppedFlag, kSFA_CUTStoppedFlag, kSFA_MeasurementStartedFlag, and kSFA_ReferenceCounterTimeOutFlag, each counter will also be cleared.

Parameters

- base – SFA peripheral base address.
- mask – SFA status flag mask (see _sfa_status_flags for bit definition).

static inline void SFA_EnableInterrupts(SFA_Type *base, uint32_t mask)

Enable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to enable (see `_sfa_interrupts_enable` for definition).

static inline void SFA_DisableInterrupts(SFA_Type *base, uint32_t mask)

Disable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to disable (see `_sfa_interrupts_enable` for definition).

static inline uint8_t SFA_GetMode(SFA_Type *base)

Get SFA measurement mode.

Parameters

- base – SFA peripheral base address.

static inline uint8_t SFA_GetCUTPredivide(SFA_Type *base)

Get CUT predivide value.

Parameters

- base – SFA peripheral base address.

void SFA_InstallCallback(SFA_Type *base, *sfa_callback_t* function)

Install the callback function to be called when IRQ happens or measurement completes.

Parameters

- base – SFA peripheral base address.
- function – the SFA measure completed callback function.

void SFA_SetMeasureConfig(SFA_Type *base, const *sfa_config_t* *config)

Set Measurement options with the passed in configuration structure.

Parameters

- base – SFA peripheral base address.
- config – SFA configuration structure.

status_t SFA_MeasureBlocking(SFA_Type *base)

Start SFA measurement in blocking mode.

Parameters

- base – SFA peripheral base address.

Return values

- `kStatus_SFA_MeasurementCompleted` – SFA measure completes.
- `kStatus_SFA_ReferenceCounterTimeout` – reference counter timeout error happens.
- `kStatus_SFA_CUTCounterTimeout` – CUT counter time out happens.

`void SFA_MeasureNonBlocking(SFA_Type *base)`

Start measure sequence in NonBlocking mode.

This function performs nonblocking measurement by enabling sfa interrupt (Please enable the `FreqGreaterThanMax` and `FreqLessThanMin` interrupts individually as needed). The callback function must be installed before invoking this function.

Note: This function has different functions for different instances.

Parameters

- `base` – SFA peripheral base address.

`void SFA_AbortMeasureSequence(SFA_Type *base)`

Abort SFA measurement sequence.

Parameters

- `base` – SFA peripheral base address.

`uint32_t SFA_CalculateFrequencyOrPeriod(SFA_Type *base, uint32_t refFrequency)`

Calculate the frequency or period.

Parameters

- `base` – SFA peripheral base address.
- `refFrequency` – The reference clock frequency(BUS clock recommended).

`static inline uint32_t SFA_GetCUTCOUNTER(SFA_Type *base)`

Get current count of the clock under test.

Parameters

- `base` – SFA peripheral base address.

`static inline void SFA_SetCUTTargetCount(SFA_Type *base, uint32_t count)`

Set the target count for the clock under test.

Parameters

- `base` – SFA peripheral base address.
- `count` – target count for CUT.

`static inline uint32_t SFA_GetCUTTargetCount(SFA_Type *base)`

Get the target count of the clock under test.

Parameters

- `base` – SFA peripheral base address.

`static inline void SFA_SetCUTLowLimitClockCount(SFA_Type *base, uint32_t count)`

Set CUT low limit clock count.

Parameters

- `base` – SFA peripheral base address.
- `count` – low limit count for CUT clock.

`static inline uint32_t SFA_GetCUTLowLimitClockCount(SFA_Type *base)`

Get CUT low limit clock count.

Parameters

- `base` – SFA peripheral base address.

static inline void SFA_SetCUTHighLimitClockCount(SFA_Type *base, uint32_t count)
Set CUT high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for CUT clock.

static inline uint32_t SFA_GetCUTHighLimitClockCount(SFA_Type *base)
Get CUT high limit clock count.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFCounter(SFA_Type *base)
Get current count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFTargetCount(SFA_Type *base, uint32_t count)
Set the target count for the reference clock.

Parameters

- base – SFA peripheral base address.
- count – target count for reference clock.

static inline uint32_t SFA_GetREFTargetCount(SFA_Type *base)
Get the target count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFStartCount(SFA_Type *base)
Get saved reference clock counter which is loaded when measurement start.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFEndCount(SFA_Type *base)
Get saved reference clock counter which is loaded when measurement complete.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFLowLimitClockCount(SFA_Type *base, uint32_t count)
Set REF low limit clock count.

Parameters

- base – SFA peripheral base address.
- count – low limit count for REF clock.

static inline uint32_t SFA_GetREFLowLimitClockCount(SFA_Type *base)
Get REF low limit clock count.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFHighLimitClockCount(SFA_Type *base, uint32_t count)

Set REF high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for REF clock.

static inline uint32_t SFA_GetREFHighLimitClockCount(SFA_Type *base)

Get REF high limit clock count.

Parameters

- base – SFA peripheral base address.

FSL_SFA_DRVIER_VERSION

SFA driver version 2.1.3.

SFA_MEASUREMENT_START_TIMEOUT

Max loops to wait for SFA measurement started.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_CUT_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA CUT counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_REF_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA REF counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA status return codes.

enumeration _sfa_status

Values:

enumerator kStatus_SFA_MeasurementCompleted

Measurement completed

enumerator kStatus_SFA_ReferenceCounterTimeout

Reference counter timeout

enumerator kStatus_SFA_CUTCounterTimeout

CUT counter timeout

enumerator kStatus_SFA_CUTClockFreqLessThanMinLimit

CUT clock frequency less than minimum limit

enumerator kStatus_SFA_CUTClockFreqGreaterThanMaxLimit

CUT clock frequency greater than maximum limit

enum _sfa_status_flags

List of SFA status flags.

The following status register flags can be cleared on any write to REF_CNT.

- kSFA_RefStoppedFlag
- kSFA_CutStoppedFlag

- kSFA_MeasurementStartedFlag
- kSFA_ReferenceCounterTimeOutFlag

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kSFA_RefStoppedFlag

Reference counter stopped flag

enumerator kSFA_CutStoppedFlag

CUT counter stopped flag

enumerator kSFA_MeasurementStartedFlag

Measurement Started flag

enumerator kSFA_ReferenceCounterTimeOutFlag

Reference counter time out flag

enumerator kSFA_InterruptRequestFlag

SFA interrupt request flag

enumerator kSFA_FreqGreaterThanMaxInterruptFlag

FREQ_GT_MAX interrupt flag

enumerator kSFA_FreqLessThanMinInterruptFlag

FREQ_LT_MIN interrupt flag

enumerator kSFA_AllStatusFlags

enum _sfa_interrupts_enable

List of SFA interrupt.

Values:

enumerator kSFA_InterruptEnable

SFA interrupt enable

enumerator kSFA_FreqGreaterThanMaxInterruptEnable

FREQ_GT_MAX interrupt enable

enumerator kSFA_FreqLessThanMinInterruptEnable

FREQ_LT_MIN interrupt enable

enum _sfa_measurement_mode

List of SFA measurement mode(Please check the mode configuration according to the manual).

Values:

enumerator kSFA_FrequencyMeasurement0

Frequency measurement performed with REF frequency > CUT frequency

enumerator kSFA_FrequencyMeasurement1

Frequency measurement performed with REF frequency < CUT frequency

enumerator kSFA_CUTPeriodMeasurement

CUT period measurement performed

enumerator kSFA_TriggerBasedMeasurement

Trigger based measurement performed

enum _sfa_cut_select

List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

Values:

enumerator kSFA__CUTSelect0

enumerator kSFA__CUTSelect1

enumerator kSFA__CUTSelect2

enumerator kSFA__CUTSelect3

enumerator kSFA__CUTSelect4

enumerator kSFA__CUTSelect5

enumerator kSFA__CUTSelect6

enumerator kSFA__CUTSelect7

enumerator kSFA__CUTSelect8

enumerator kSFA__CUTSelect9

enumerator kSFA__CUTSelect10

enumerator kSFA__CUTSelect11

enumerator kSFA__CUTSelect12

enumerator kSFA__CUTSelect13

enumerator kSFA__CUTSelect14

enumerator kSFA__CUTSelect15

enum _sfa_ref_select

List of REF which is connected to the REF counter (Please refer to the manual for configuration).

Values:

enumerator kSFA__REFSelect0

enumerator kSFA__REFSelect1

enumerator kSFA__REFSelect2

enum _sfa_trigger_start_select

List of Signal MUX for Trigger Based Measurement Start.

Values:

enumerator kSFA__TriggerStartSelect0

enumerator kSFA__TriggerStartSelect1

enum _sfa_trigger_end_select

List of Signal MUX for Trigger Based Measurement End.

Values:

enumerator kSFA__TriggerEndSelect0

enumerator kSFA_TriggerEndSelect1

enum _sfa_trigger_start_polarity

List of Trigger Start Polarity.

Values:

enumerator kSFA_TriggerStartPolarityRiseEdge

Rising edge will begin the measurement sequence

enumerator kSFA_TriggerStartPolarityFallEdge

Falling edge will begin the measurement sequence

enum _sfa_trigger_end_polarity

List of Trigger End Polarity.

Values:

enumerator kSFA_TriggerEndPolarityRiseEdge

Rising edge will end the measurement sequence

enumerator kSFA_TriggerEndPolarityFallEdge

Falling edge will end the measurement sequence

typedef void (*sfa_callback_t)(status_t status)

sfa measure completion callback function pointer type

This callback can be used in non blocking IRQHandler. Specify the callback you want in the call to SFA_InstallCallback().

Param base

SFA peripheral base address.

Param status

The runtime measurement status. kStatus_SFA_MeasurementCompleted: The measurement completes. kStatus_SFA_ReferenceCounterTimeout: Reference counter timeout happens. kStatus_SFA_CUTCounterTimeout: CUT counter timeout happens.

typedef enum _sfa_measurement_mode sfa_measurement_mode_t

List of SFA measurement mode(Please check the mode configuration according to the manual).

typedef enum _sfa_cut_select sfa_cut_select_t

List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

typedef enum _sfa_ref_select sfa_ref_select_t

List of REF which is connected to the REF counter (Please refer to the manual for configuration).

typedef enum _sfa_trigger_start_select sfa_trigger_start_select_t

List of Signal MUX for Trigger Based Measurement Start.

typedef enum _sfa_trigger_end_select sfa_trigger_end_select_t

List of Signal MUX for Trigger Based Measurement End.

typedef enum _sfa_trigger_start_polarity sfa_trigger_start_polarity_t

List of Trigger Start Polarity.

typedef enum _sfa_trigger_end_polarity sfa_trigger_end_polarity_t

List of Trigger End Polarity.

```
typedef struct _sfa_init_config sfa_config_t
```

Structure with setting to initialize the SFA module.

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

```
SFA__CUT__CLK__Enable(val)
```

```
struct _sfa_init_config
```

```
#include <fsl_sfa.h> Structure with setting to initialize the SFA module.
```

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

Public Members

```
sfa_measurement_mode_t mode
```

measurement mode

```
sfa_cut_select_t cutSelect
```

Select clock connected to the clock under test counter

```
sfa_ref_select_t refSelect
```

Select REF connected the bus clock

```
uint8_t prediv
```

Integer divide of the Input CUT signal

```
sfa_trigger_start_select_t trigStart
```

Select the signal will be used to end a trigger based measurement

```
sfa_trigger_start_polarity_t startPolarity
```

Select the polarity of the start trigger signal

```
sfa_trigger_end_select_t trigEnd
```

Select the signal will be used to commence a trigger based measurement

```
sfa_trigger_end_polarity_t endPolarity
```

Select the polarity of the end trigger signal

```
bool enableTrigMeasurement
```

false: The measurement will start by default with a dummy write to the CUT counter;
true : The measurement will start after receiving a dummy write to the REF_CNT followed by receiving the trigger edge

```
bool enableCUTPin
```

Control the connection of the clock under test to an external pin.

```
uint32_t refTarget
```

Reference counter target counts

```
uint32_t cutTarget
```

CUT counter target counts

2.71 SIM: System Integration Module Driver

FSL_SIM_DRIVER_VERSION

Driver version.

typedef struct *_sim_uid* *sim_uid_t*

Unique ID.

void SIM_GetUniqueId(*sim_uid_t* *uid)

Gets the unique identification register value.

Parameters

- uid – Pointer to the structure to save the UID value.

struct *_sim_uid*

#include <*fsl_sim.h*> Unique ID.

Public Members

uint32_t MH

UIDMH.

uint32_t ML

UIDML.

uint32_t L

UIDL.

2.72 Soc_mipi_dsi

FSL_SOC_MIPI_DSI_DRIVER_VERSION

Driver version.

DSI_DPHY_PLL_VCO_MAX

DSI_DPHY_PLL_VCO_MIN

FSL_COMPONENT_ID

2.73 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)

Tx 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_ValidityNoGood
SPDIF validity 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

2.74 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

2.75 SYSPM: System Performance Monitor

enum __syspm_monitor
syspm select control monitor
Values:
enumerator kSYSPM_Monitor0
Monitor 0

enum __syspm_event
syspm select event
Values:
enumerator kSYSPM_Event1
Event 1
enumerator kSYSPM_Event2
Event 2

enumerator kSYSPM_Event3
Event 3

enum __syspm__mode

syspm set count mode

Values:

enumerator kSYSPM_BothMode
count in both modes

enumerator kSYSPM_UserMode
count only in user mode

enumerator kSYSPM_PrivilegedMode
count only in privileged mode

enum __syspm__startstop__control

syspm start/stop control

Values:

enumerator kSYSPM_Idle
idle >

enumerator kSYSPM_LocalStop
local stop

enumerator kSYSPM_LocalStart
local start

enumerator kSYSPM_EnableTraceControl
enable global TSTART/TSTOP

enumerator kSYSPM_GlobalStart
global stop

enumerator kSYSPM_GlobalStop
global start

typedef enum __syspm__monitor syspm_monitor_t

syspm select control monitor

typedef enum __syspm__event syspm_event_t

syspm select event

typedef enum __syspm__mode syspm_mode_t

syspm set count mode

typedef enum __syspm__startstop__control syspm_startstop_control_t

syspm start/stop control

void SYSPM_Init(SYSPM_Type *base)

Initializes the SYSPM.

This function enables the SYSPM clock.

Parameters

- base – SYSPM peripheral base address.

void SYSPM_Deinit(SYSPM_Type *base)

Deinitializes the SYSPM.

This function disables the SYSPM clock.

Parameters

- base – SYSPM peripheral base address.

```
void SYSPM_SelectEvent(SYSPM_Type *base, syspm_monitor_t monitor, syspm_event_t event,
                      uint8_t eventCode)
```

Select event counters.

Parameters

- base – SYSPM peripheral base address.
- event – syspm select event, see to syspm_event_t.
- eventCode – select which event to be counted in PMECTR_x, see to table Events.

```
void SYSPM_ResetEvent(SYSPM_Type *base, syspm_monitor_t monitor, syspm_event_t event)
```

Reset event counters.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.

```
void SYSPM_ResetInstructionEvent(SYSPM_Type *base, syspm_monitor_t monitor)
```

Reset Instruction Counter.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.

```
void SYSPM_SetCountMode(SYSPM_Type *base, syspm_monitor_t monitor, syspm_mode_t
                       mode)
```

Set count mode.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- mode – syspm select counter mode, see to syspm_mode_t.

```
void SYSPM_SetStartStopControl(SYSPM_Type *base, syspm_monitor_t monitor,
                              syspm_startstop_control_t ssc)
```

Set Start/Stop Control.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- ssc – This 3-bit field provides a three-phase mechanism to start/stop the counters. It includes a prioritized scheme with local start > local stop > global start > global stop > conditional TSTART > TSTOP. The global and conditional start/stop affect all configured PM/PSAM module concurrently so counters are “coherent”. see to syspm_startstop_control_t

```
void SYSPM_DisableCounter(SYSPM_Type *base, syspm_monitor_t monitor)
```

Disable Counters if Stopped or Halted.

Parameters

- base – SYSPM peripheral base address.

- monitor – syspm control monitor, see to syspm_monitor_t.

uint64_t SYSPM_GetEventCounter(SYSpm_Type *base, syspm_monitor_t monitor, syspm_event_t event)

This is the the 40-bits of eventx counter. The value in this register increments each time the event selected in PMCRx[SELEVTx] occurs.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- event – syspm select event, see to syspm_event_t.

Returns

- When the return value is not equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents a 40 bits eventx counter.
- When the return value is equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents timeout occurred.

EVENT_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM event count stable (0 means wait forever)

INSTRUCTION_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM instruction count stable (0 means wait forever)

FSL_SYSPM_DRIVER_VERSION

SYSPM driver version.

SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE

2.76 TPM: Timer PWM Module

uint32_t TPM_GetInstance(TPM_Type *base)

Gets the instance from the base address.

Parameters

- base – TPM peripheral base address

Returns

The TPM instance

void TPM_Init(TPM_Type *base, const tpm_config_t *config)

Ungates the TPM clock and configures the peripheral for basic operation.

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

Parameters

- base – TPM peripheral base address
- config – Pointer to user's TPM config structure.

void TPM_Deinit(TPM_Type *base)

Stops the counter and gates the TPM clock.

Parameters

- base – TPM peripheral base address

void TPM_GetDefaultConfig(*tpm_config_t* *config)

Fill in the TPM config struct with the default settings.

The default values are:

```
config->prescale = kTPM_Prescale_Divide_1;
config->useGlobalTimeBase = false;
config->syncGlobalTimeBase = false;
config->dozeEnable = false;
config->dbgMode = false;
config->enableReloadOnTrigger = false;
config->enableStopOnOverflow = false;
config->enableStartOnTrigger = false;
#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
config->enablePauseOnTrigger = false;
#endif
config->triggerSelect = kTPM_Trigger_Select_0;
#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
config->triggerSource = kTPM_TriggerSource_External;
config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
#endif
#if defined(FSL_FEATURE_TPM_HAS_POL) && FSL_FEATURE_TPM_HAS_POL
config->chnlPolarity = 0U;
#endif
```

Parameters

- config – Pointer to user's TPM config structure.

tpm_clock_prescale_t TPM_CalculateCounterClkDiv(TPM_Type *base, uint32_t counterPeriod_Hz, uint32_t srcClock_Hz)

Calculates the counter clock prescaler.

This function calculates the values for SC[PS].

return Calculated clock prescaler value.

Parameters

- base – TPM peripheral base address
- counterPeriod_Hz – The desired frequency in Hz which corresponding to the time when the counter reaches the mod value
- srcClock_Hz – TPM counter clock in Hz

status_t TPM_SetupPwm(TPM_Type *base, const *tpm_chnl_pwm_signal_param_t* *chnlParams, uint8_t numOfChnls, *tpm_pwm_mode_t* mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)

Configures the PWM signal parameters.

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

- base – TPM peripheral base address
- chnlParams – Array of PWM channel parameters to configure the channel(s)
- numOfChnls – Number of channels to configure, this should be the size of the array passed in

- mode – PWM operation mode, options available in enumeration `tpm_pwm_mode_t`
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – TPM counter clock in Hz

Returns

`kStatus_Success` PWM setup successful
`kStatus_Error` PWM setup failed
`kStatus_Timeout` PWM setup timeout when write register CnV or MOD

`status_t` TPM_UpdatePwmDutycycle(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_pwm_mode_t* currentPwmMode, uint8_t dutyCyclePercent)

Update the duty cycle of an active PWM signal.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number. In combined mode, this represents the channel pair number
- currentPwmMode – The current PWM mode set during PWM setup
- dutyCyclePercent – New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

Returns

`kStatus_Success` if the PWM setup was successful, `kStatus_Error` on failure

`void` TPM_UpdateChnlEdgeLevelSelect(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t level)

Update the edge level selection for a channel.

Note: When the TPM has PWM pause level select feature (FSL_FEATURE_TPM_HAS_PAUSE_LEVEL_SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use TPM_DisableChannel API to close the PWM output.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- level – The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field.

`static inline uint8_t` TPM_GetChannelControlBits(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Get the channel control bits value (mode, edge and level bit fields).

This function disable the channel by clear all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The control bits value. This is the logical OR of members of the enumeration `tpm_chnl_control_bit_mask_t`.

```
static inline status_t TPM_DisableChannel(TPM_Type *base, tpm_chnl_t chnlNumber)
```

Disable the channel.

This function disable the channel by clear all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

```
static inline status_t TPM_EnableChannel(TPM_Type *base, tpm_chnl_t chnlNumber, uint8_t control)
```

Enable the channel according to mode and level configs.

This function enable the channel output according to input mode/level config parameters.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- control – The control bits value. This is the logical OR of members of the enumeration *tpm_chnl_control_bit_mask_t*.

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

```
void TPM_SetupInputCapture(TPM_Type *base, tpm_chnl_t chnlNumber, tpm_input_capture_edge_t captureMode)
```

Enables capturing an input signal on the channel using the function parameters.

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- captureMode – Specifies which edge to capture

```
status_t TPM_SetupOutputCompare(TPM_Type *base, tpm_chnl_t chnlNumber, tpm_output_compare_mode_t compareMode, uint32_t compareValue)
```

Configures the TPM to generate timed pulses.

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- compareMode – Action to take on the channel output when the compare condition is met
- compareValue – Value to be programmed in the CnV register.

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout
when write register CnV

```
void TPM_SetupDualEdgeCapture(TPM_Type *base, tpm_chnl_t chnlPairNumber, const  
                             tpm_dual_edge_capture_param_t *edgeParam, uint32_t  
                             filterValue)
```

Configures the dual edge capture mode of the TPM.

This function allows to measure a pulse width of the signal on the input of channel of a channel pair. The filter function is disabled if the filterVal argument passed is zero.

Parameters

- base – TPM peripheral base address
- chnlPairNumber – The TPM channel pair number; options are 0, 1, 2, 3
- edgeParam – Sets up the dual edge capture function
- filterValue – Filter value, specify 0 to disable filter.

```
void TPM_SetupQuadDecode(TPM_Type *base, const tpm_phase_params_t *phaseAParams,  
                        const tpm_phase_params_t *phaseBParams,  
                        tpm_quad_decode_mode_t quadMode)
```

Configures the parameters and activates the quadrature decode mode.

Parameters

- base – TPM peripheral base address
- phaseAParams – Phase A configuration parameters
- phaseBParams – Phase B configuration parameters
- quadMode – Selects encoding mode used in quadrature decoder mode

```
static inline void TPM_SetChannelPolarity(TPM_Type *base, tpm_chnl_t chnlNumber, bool  
                                         enable)
```

Set the input and output polarity of each of the channels.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- enable – true: Set the channel polarity to active high; false: Set the channel polarity to active low;

```
static inline void TPM_EnableChannelExtTrigger(TPM_Type *base, tpm_chnl_t chnlNumber, bool  
                                              enable)
```

Enable external trigger input to be used by channel.

In input capture mode, configures the trigger input that is used by the channel to capture the counter value. In output compare or PWM mode, configures the trigger input used to modulate the channel output. When modulating the output, the output is forced to the channel initial value whenever the trigger is not asserted.

Note: No matter how many external trigger sources there are, only input trigger 0 and 1 are used. The even numbered channels share the input trigger 0 and the odd numbered channels share the second input trigger 1.

Parameters

- base – TPM peripheral base address

- `chnlNumber` – The channel number
- `enable` – `true`: Configures trigger input 0 or 1 to be used by channel; `false`: Trigger input has no effect on the channel

`void TPM_EnableInterrupts(TPM_Type *base, uint32_t mask)`

Enables the selected TPM interrupts.

Parameters

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

`void TPM_DisableInterrupts(TPM_Type *base, uint32_t mask)`

Disables the selected TPM interrupts.

Parameters

- `base` – TPM peripheral base address
- `mask` – The interrupts to disable. This is a logical OR of members of the enumeration `tpm_interrupt_enable_t`

`uint32_t TPM_GetEnabledInterrupts(TPM_Type *base)`

Gets the enabled TPM interrupts.

Parameters

- `base` – TPM peripheral base address

Returns

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

`void TPM_RegisterCallBack(TPM_Type *base, tpm_callback_t callback)`

Register callback.

If channel or overflow interrupt is enabled by the user, then a callback can be registered which will be invoked when the interrupt is triggered.

Parameters

- `base` – TPM peripheral base address
- `callback` – Callback function

`static inline uint32_t TPM_GetChannelValue(TPM_Type *base, tpm_chnl_t chnlNumber)`

Gets the TPM channel value.

Note: The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

- `base` – TPM peripheral base address
- `chnlNumber` – The channel number

Returns

The channle CnV regisyer value.

`static inline uint32_t TPM_GetStatusFlags(TPM_Type *base)`

Gets the TPM status flags.

Parameters

- base – TPM peripheral base address

Returns

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

```
static inline void TPM_ClearStatusFlags(TPM_Type *base, uint32_t mask)
```

Clears the TPM status flags.

Parameters

- base – TPM peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `tpm_status_flags_t`

```
static inline status_t TPM_SetTimerPeriod(TPM_Type *base, uint32_t ticks)
```

Sets the timer period in units of ticks.

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note:

- a. This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.
 - b. Call the utility macros provided in the `fsl_common.h` to convert usec or msec to ticks.
-

Parameters

- base – TPM peripheral base address
- ticks – A timer period in units of ticks, which should be equal or greater than 1.

Returns

`kStatus_Success` PWM setup successful `kStatus_Timeout` PWM setup timeout when write register `CnSC`

```
static inline uint32_t TPM_GetCurrentTimerCount(TPM_Type *base)
```

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – TPM peripheral base address

Returns

The current counter value in ticks

```
static inline void TPM_StartTimer(TPM_Type *base, tpm_clock_source_t clockSource)
```

Starts the TPM counter.

Parameters

- base – TPM peripheral base address
- clockSource – TPM clock source; once clock source is set the counter will start running

static inline *status_t* TPM_StopTimer(TPM_Type *base)

Stops the TPM counter.

Parameters

- base – TPM peripheral base address

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout
when write register CnSC

FSL_TPM_DRIVER_VERSION

TPM driver version 2.4.0.

enum *_tpm_chnl*

List of TPM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kTPM_Chnl_0
TPM channel number 0

enumerator kTPM_Chnl_1
TPM channel number 1

enumerator kTPM_Chnl_2
TPM channel number 2

enumerator kTPM_Chnl_3
TPM channel number 3

enumerator kTPM_Chnl_4
TPM channel number 4

enumerator kTPM_Chnl_5
TPM channel number 5

enumerator kTPM_Chnl_6
TPM channel number 6

enumerator kTPM_Chnl_7
TPM channel number 7

enum *_tpm_pwm_mode*

TPM PWM operation modes.

Values:

enumerator kTPM_EdgeAlignedPwm
Edge aligned PWM

enumerator kTPM_CenterAlignedPwm
Center aligned PWM

enumerator kTPM_CombinedPwm
Combined PWM (Edge-aligned, center-aligned, or asymmetrical PWMs can be obtained in combined mode using different software configurations)

enum _tpm_pwm_level_select

TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Values:

enumerator kTPM_HighTrue

High true pulses

enumerator kTPM_LowTrue

Low true pulses

enum _tpm_pwm_pause_level_select

TPM PWM output when first enabled or paused: set or clear.

Values:

enumerator kTPM_ClearOnPause

Clear Output when counter first enabled or paused.

enumerator kTPM_SetOnPause

Set Output when counter first enabled or paused.

enum _tpm_chnl_control_bit_mask

List of TPM channel modes and level control bit mask.

Values:

enumerator kTPM_ChnlELSnAMask

Channel ELSA bit mask.

enumerator kTPM_ChnlELSnBMask

Channel ELSE bit mask.

enumerator kTPM_ChnlMSAMask

Channel MSA bit mask.

enumerator kTPM_ChnlMSBMask

Channel MSB bit mask.

enum _tpm_trigger_select

Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

Values:

enumerator kTPM_Trigger_Select_0

enumerator kTPM_Trigger_Select_1

enumerator kTPM_Trigger_Select_2

enumerator kTPM_Trigger_Select_3

enumerator kTPM_Trigger_Select_4
enumerator kTPM_Trigger_Select_5
enumerator kTPM_Trigger_Select_6
enumerator kTPM_Trigger_Select_7
enumerator kTPM_Trigger_Select_8
enumerator kTPM_Trigger_Select_9
enumerator kTPM_Trigger_Select_10
enumerator kTPM_Trigger_Select_11
enumerator kTPM_Trigger_Select_12
enumerator kTPM_Trigger_Select_13
enumerator kTPM_Trigger_Select_14
enumerator kTPM_Trigger_Select_15

enum _tpm_trigger_source

Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

Values:

enumerator kTPM_TriggerSource_External
 Use external trigger input
enumerator kTPM_TriggerSource_Internal
 Use internal trigger (channel pin input capture)

enum _tpm_ext_trigger_polarity

External trigger source polarity.

Note: Selects the polarity of the external trigger source.

Values:

enumerator kTPM_ExtTrigger_Active_High
 External trigger input is active high
enumerator kTPM_ExtTrigger_Active_Low
 External trigger input is active low

enum _tpm_output_compare_mode

TPM output compare modes.

Values:

enumerator kTPM_NoOutputSignal
 No channel output when counter reaches CnV
enumerator kTPM_ToggleOnMatch
 Toggle output

enumerator kTPM_ClearOnMatch

Clear output

enumerator kTPM_SetOnMatch

Set output

enumerator kTPM_HighPulseOutput

Pulse output high

enumerator kTPM_LowPulseOutput

Pulse output low

enum _tpm_input_capture_edge

TPM input capture edge.

Values:

enumerator kTPM_RisingEdge

Capture on rising edge only

enumerator kTPM_FallingEdge

Capture on falling edge only

enumerator kTPM_RiseAndFallEdge

Capture on rising or falling edge

enum _tpm_quad_decode_mode

TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

Values:

enumerator kTPM_QuadPhaseEncode

Phase A and Phase B encoding mode

enumerator kTPM_QuadCountAndDir

Count and direction encoding mode

enum _tpm_phase_polarity

TPM quadrature phase polarities.

Values:

enumerator kTPM_QuadPhaseNormal

Phase input signal is not inverted

enumerator kTPM_QuadPhaseInvert

Phase input signal is inverted

enum _tpm_clock_source

TPM clock source selection.

Values:

enumerator kTPM_SystemClock

System clock

enumerator kTPM_ExternalClock

External TPM_EXTCLK pin clock

enumerator kTPM_ExternalInputTriggerClock

Selected external input trigger clock

enum _tpm_clock_prescale

TPM prescale value selection for the clock source.

Values:

enumerator kTPM_Prescale_Divide_1

Divide by 1

enumerator kTPM_Prescale_Divide_2

Divide by 2

enumerator kTPM_Prescale_Divide_4

Divide by 4

enumerator kTPM_Prescale_Divide_8

Divide by 8

enumerator kTPM_Prescale_Divide_16

Divide by 16

enumerator kTPM_Prescale_Divide_32

Divide by 32

enumerator kTPM_Prescale_Divide_64

Divide by 64

enumerator kTPM_Prescale_Divide_128

Divide by 128

enum _tpm_interrupt_enable

List of TPM interrupts.

Values:

enumerator kTPM_Chnl0InterruptEnable

Channel 0 interrupt.

enumerator kTPM_Chnl1InterruptEnable

Channel 1 interrupt.

enumerator kTPM_Chnl2InterruptEnable

Channel 2 interrupt.

enumerator kTPM_Chnl3InterruptEnable

Channel 3 interrupt.

enumerator kTPM_Chnl4InterruptEnable

Channel 4 interrupt.

enumerator kTPM_Chnl5InterruptEnable

Channel 5 interrupt.

enumerator kTPM_Chnl6InterruptEnable

Channel 6 interrupt.

enumerator kTPM_Chnl7InterruptEnable

Channel 7 interrupt.

enumerator kTPM_TimeOverflowInterruptEnable

Time overflow interrupt.

enum *_tpm_status_flags*

List of TPM flags.

Values:

enumerator *kTPM_Chnl0Flag*

Channel 0 flag

enumerator *kTPM_Chnl1Flag*

Channel 1 flag

enumerator *kTPM_Chnl2Flag*

Channel 2 flag

enumerator *kTPM_Chnl3Flag*

Channel 3 flag

enumerator *kTPM_Chnl4Flag*

Channel 4 flag

enumerator *kTPM_Chnl5Flag*

Channel 5 flag

enumerator *kTPM_Chnl6Flag*

Channel 6 flag

enumerator *kTPM_Chnl7Flag*

Channel 7 flag

enumerator *kTPM_TimeOverflowFlag*

Time overflow flag

typedef enum *_tpm_chnl* *tpm_chnl_t*

List of TPM channels.

Note: Actual number of available channels is SoC dependent

typedef enum *_tpm_pwm_mode* *tpm_pwm_mode_t*

TPM PWM operation modes.

typedef enum *_tpm_pwm_level_select* *tpm_pwm_level_select_t*

TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

typedef enum *_tpm_pwm_pause_level_select* *tpm_pwm_pause_level_select_t*

TPM PWM output when first enabled or paused: set or clear.

typedef enum *_tpm_chnl_control_bit_mask* *tpm_chnl_control_bit_mask_t*

List of TPM channel modes and level control bit mask.

typedef struct *_tpm_chnl_pwm_signal_param* *tpm_chnl_pwm_signal_param_t*

Options to configure a TPM channel's PWM signal.

`typedef enum _tpm_trigger_select tpm_trigger_select_t`

Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

`typedef enum _tpm_trigger_source tpm_trigger_source_t`

Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

`typedef enum _tpm_ext_trigger_polarity tpm_ext_trigger_polarity_t`

External trigger source polarity.

Note: Selects the polarity of the external trigger source.

`typedef enum _tpm_output_compare_mode tpm_output_compare_mode_t`

TPM output compare modes.

`typedef enum _tpm_input_capture_edge tpm_input_capture_edge_t`

TPM input capture edge.

`typedef struct _tpm_dual_edge_capture_param tpm_dual_edge_capture_param_t`

TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

`typedef enum _tpm_quad_decode_mode tpm_quad_decode_mode_t`

TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

`typedef enum _tpm_phase_polarity tpm_phase_polarity_t`

TPM quadrature phase polarities.

`typedef struct _tpm_phase_param tpm_phase_params_t`

TPM quadrature decode phase parameters.

`typedef enum _tpm_clock_source tpm_clock_source_t`

TPM clock source selection.

`typedef enum _tpm_clock_prescale tpm_clock_prescale_t`

TPM prescale value selection for the clock source.

`typedef struct _tpm_config tpm_config_t`

TPM config structure.

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the `TPM_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

`typedef enum _tpm_interrupt_enable tpm_interrupt_enable_t`

List of TPM interrupts.

`typedef enum _tpm_status_flags tpm_status_flags_t`

List of TPM flags.

`typedef void (*tpm_callback_t)(TPM_Type *base)`

TPM callback function pointer.

Param base

TPM peripheral base address.

`static inline void TPM_Reset(TPM_Type *base)`

Performs a software reset on the TPM module.

Reset all internal logic and registers, except the Global Register. Remains set until cleared by software.

Note: TPM software reset is available on certain SoC's only

Parameters

- base – TPM peripheral base address

`void TPM_DriverIRQHandler(uint32_t instance)`

TPM driver IRQ handler common entry.

This function provides the common IRQ request entry for TPM.

Parameters

- instance – TPM instance.

`TPM_TIMEOUT`

Max loops to wait for writing register.

When writing MOD CnV CnSC and SC register, driver will wait until register is updated. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

`TPM_MAX_COUNTER_VALUE(x)`

Help macro to get the max counter value.

`struct _tpm_chnl_pwm_signal_param`

`#include <fsl_tpm.h>` Options to configure a TPM channel's PWM signal.

Public Members

`tpm_chnl_t` chnlNumber

TPM channel to configure. In combined mode (available in some SoC's), this represents the channel pair number

`tpm_pwm_pause_level_select_t` pauseLevel

PWM output level when counter first enabled or paused

`tpm_pwm_level_select_t` level

PWM output active level select

`uint8_t` dutyCyclePercent

PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=always active signal (100% duty cycle)

uint8_t firstEdgeDelayPercent

Used only in combined PWM mode to generate asymmetrical PWM. Specifies the delay to the first edge in a PWM period. If unsure, leave as 0. Should be specified as percentage of the PWM period, (dutyCyclePercent + firstEdgeDelayPercent) value should be not greater than 100.

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

tpm_pwm_pause_level_select_t secPauseLevel

Used only in combined PWM mode. Define the second channel output level when counter first enabled or paused

uint8_t deadTimeValue[2]

The dead time value for channel n and n+1 in combined complementary PWM mode. Deadtime insertion is disabled when this value is zero, otherwise deadtime insertion for channel n/n+1 is configured as (deadTimeValue * 4) clock cycles. deadTimeValue's available range is 0 ~ 15.

struct __tpm_dual_edge_capture_param

#include <fsl_tpm.h> TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

Public Members

bool enableSwap

true: Use channel n+1 input, channel n input is ignored; false: Use channel n input, channel n+1 input is ignored

tpm_input_capture_edge_t currChanEdgeMode

Input capture edge select for channel n

tpm_input_capture_edge_t nextChanEdgeMode

Input capture edge select for channel n+1

struct __tpm_phase_param

#include <fsl_tpm.h> TPM quadrature decode phase parameters.

Public Members

uint32_t phaseFilterVal

Filter value, filter is disabled when the value is zero

tpm_phase_polarity_t phasePolarity

Phase polarity

struct __tpm_config

#include <fsl_tpm.h> TPM config structure.

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

The config struct can be made const so it resides in flash

Public Members

tpm_clock_prescale_t prescale

Select TPM clock prescale value

bool useGlobalTimeBase

true: The TPM channels use an external global time base (the local counter still use for generate overflow interrupt and DMA request); false: All TPM channels use the local counter as their timebase

bool syncGlobalTimeBase

true: The TPM counter is synchronized to the global time base; false: disabled

tpm_trigger_select_t triggerSelect

Input trigger to use for controlling the counter operation

tpm_trigger_source_t triggerSource

Decides if we use external or internal trigger.

tpm_ext_trigger_polarity_t extTriggerPolarity

when using external trigger source, need selects the polarity of it.

bool enableDoze

true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode

bool enableDebugMode

true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode

bool enableReloadOnTrigger

true: TPM counter is reloaded on trigger; false: TPM counter not reloaded

bool enableStopOnOverflow

true: TPM counter stops after overflow; false: TPM counter continues running after overflow

bool enableStartOnTrigger

true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately

bool enablePauseOnTrigger

true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running

uint8_t chnlPolarity

Defines the input/output polarity of the channels in POL register

2.77 TRDC: Trusted Resource Domain Controller

void TRDC_Init(TRDC_Type *base)

Initializes the TRDC module.

This function enables the TRDC clock.

Parameters

- base – TRDC peripheral base address.

`void TRDC_Deinit(TRDC_Type *base)`

De-initializes the TRDC module.

This function disables the TRDC clock.

Parameters

- `base` – TRDC peripheral base address.

`static inline uint8_t TRDC_GetCurrentMasterDomainId(TRDC_Type *base)`

Gets the domain ID of the current bus master.

Parameters

- `base` – TRDC peripheral base address.

Returns

Domain ID of current bus master.

`void TRDC_GetHardwareConfig(TRDC_Type *base, trdc_hardware_config_t *config)`

Gets the TRDC hardware configuration.

This function gets the TRDC hardware configurations, including number of bus masters, number of domains, number of MRCs and number of PACs.

Parameters

- `base` – TRDC peripheral base address.
- `config` – Pointer to the structure to get the configuration.

`static inline void TRDC_SetDacGlobalValid(TRDC_Type *base)`

Sets the TRDC DAC(Domain Assignment Controllers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- `base` – TRDC peripheral base address.

`static inline void TRDC_LockMasterDomainAssignment(TRDC_Type *base, uint8_t master)`

Locks the bus master domain assignment register.

This function locks the master domain assignment. After it is locked, the register can't be changed until next reset.

Parameters

- `base` – TRDC peripheral base address.
- `master` – Which master to configure.

`static inline void TRDC_SetMasterDomainAssignmentValid(TRDC_Type *base, uint8_t master, bool valid)`

Sets the master domain assignment as valid or invalid.

This function sets the master domain assignment as valid or invalid.

Parameters

- `base` – TRDC peripheral base address.
- `master` – Which master to configure.
- `valid` – True to set valid, false to set invalid.

```
void TRDC_GetDefaultProcessorDomainAssignment(trdc_processor_domain_assignment_t
                                              *domainAssignment)
```

Gets the default master domain assignment for the processor bus master.

This function gets the default master domain assignment for the processor bus master. It should only be used for the processor bus masters, such as CORE0. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->domainIdSelect = kTRDC_DidMda;
assignment->lock           = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

```
void TRDC_GetDefaultNonProcessorDomainAssignment(trdc_non_processor_domain_assignment_t
                                                 *domainAssignment)
```

Gets the default master domain assignment for non-processor bus master.

This function gets the default master domain assignment for non-processor bus master. It should only be used for the non-processor bus masters, such as DMA. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->privilegeAttr  = kTRDC_ForceUser;
assignment->secureAttr     = kTRDC_ForceSecure;
assignment->bypassDomainId = 0U;
assignment->lock           = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

```
void TRDC_SetProcessorDomainAssignment(TRDC_Type *base, const
                                       trdc_processor_domain_assignment_t
                                       *domainAssignment)
```

Sets the processor bus master domain assignment.

This function sets the processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for core 0.

```
trdc_processor_domain_assignment_t processorAssignment;

TRDC_GetDefaultProcessorDomainAssignment(&processorAssignment);

processorAssignment.domainId = 0;
processorAssignment.xxx      = xxx;
TRDC_SetMasterDomainAssignment(TRDC, &processorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- domainAssignment – Pointer to the assignment structure.

```
static inline void TRDC_EnableProcessorDomainAssignment(TRDC_Type *base, bool enable)
```

Enables the processor bus master domain assignment.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

```
void TRDC_SetNonProcessorDomainAssignment(TRDC_Type *base, uint8_t master, const
                                         trdc_non_processor_domain_assignment_t
                                         *domainAssignment)
```

Sets the non-processor bus master domain assignment.

This function sets the non-processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for DMA0.

```
trdc_non_processor_domain_assignment_t nonProcessorAssignment;

TRDC_GetDefaultNonProcessorDomainAssignment(&nonProcessorAssignment);
nonProcessorAssignment.domainId = 1;
nonProcessorAssignment.xxx      = xxx;

TRDC_SetMasterDomainAssignment(TRDC, kTrdcMasterDma0, 0U, &nonProcessorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- master – Which master to configure, refer to `trdc_master_t` in processor header file.
- domainAssignment – Pointer to the assignment structure.

```
void TRDC_GetDefaultIDAUConfig(trdc_idau_config_t *idauConfiguration)
```

Gets the default IDAU(Implementation-Defined Attribution Unit) configuration.

```
config->lockSecureVTOR   = false;
config->lockNonsecureVTOR = false;
config->lockSecureMPU     = false;
config->lockNonsecureMPU  = false;
config->lockSAU           = false;
```

Parameters

- idauConfiguration – Pointer to the configuration structure.

```
void TRDC_SetIDAU(TRDC_Type *base, const trdc_idau_config_t *idauConfiguration)
```

Sets the IDAU(Implementation-Defined Attribution Unit) control configuration.

Example: Lock the secure and non-secure MPU registers.

```
trdc_idau_config_t idauConfiguration;

TRDC_GetDefaultIDAUConfig(&idauConfiguration);

idauConfiguration.lockSecureMPU = true;
idauConfiguration.lockNonsecureMPU = true;
TRDC_SetIDAU(TRDC, &idauConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- idauConfiguration – Pointer to the configuration structure.

static inline void TRDC_EnableFlashLogicalWindow(TRDC_Type *base, bool enable)

Enables/disables the FLW(flash logical window) function.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

static inline void TRDC_LockFlashLogicalWindow(TRDC_Type *base)

Locks FLW registers. Once locked the registers can not be updated until next reset.

Parameters

- base – TRDC peripheral base address.

static inline uint32_t TRDC_GetFlashLogicalWindowPbase(TRDC_Type *base)

Gets the FLW physical base address.

Parameters

- base – TRDC peripheral base address.

Returns

Physical address of the FLW function.

static inline void TRDC_GetSetFlashLogicalWindowSize(TRDC_Type *base, uint16_t size)

Sets the FLW size.

Parameters

- base – TRDC peripheral base address.
- size – Size of the FLW in unit of 32k bytes.

void TRDC_GetDefaultFlashLogicalWindowConfig(*trdc_flw_config_t* *flwConfiguration)

Gets the default FLW(Flash Logical Window) configuration.

```
config->blockCount = false;
config->arrayBaseAddr = false;
config->lock = false;
config->enable = false;
```

Parameters

- flwConfiguration – Pointer to the configuration structure.

void TRDC_SetFlashLogicalWindow(TRDC_Type *base, const *trdc_flw_config_t* *flwConfiguration)

Sets the FLW function's configuration.

```
trdc_flw_config_t flwConfiguration;

TRDC_GetDefaultIDAUConfig(&flwConfiguration);

flwConfiguration.blockCount = 32U;
flwConfiguration.arrayBaseAddr = 0xFFFFFFFF;
TRDC_SetIDAU(TRDC, &flwConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- flwConfiguration – Pointer to the configuration structure.

status_t TRDC_GetAndClearFirstDomainError(TRDC_Type *base, *trdc_domain_error_t* *error)

Gets and clears the first domain error of the current domain.

This function gets the first access violation information for the current domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.

Returns

If the access violation is captured, this function returns the *kStatus_Success*. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the *kStatus_NoData*.

status_t TRDC_GetAndClearFirstSpecificDomainError(TRDC_Type *base, *trdc_domain_error_t* *error, *uint8_t* domainId)

Gets and clears the first domain error of the specific domain.

This function gets the first access violation information for the specific domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.
- domainId – The error of which domain to get and clear.

Returns

If the access violation is captured, this function returns the *kStatus_Success*. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the *kStatus_NoData*.

static inline void TRDC_SetMrcGlobalValid(TRDC_Type *base)

Sets the TRDC MRC(Memory Region Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

static inline *uint8_t* TRDC_GetMrcRegionNumber(TRDC_Type *base, *uint8_t* mrcIdx)

Gets the TRDC MRC(Memory Region Checkers) region number valid.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.

Returns

the region number of the given MRC instance

void TRDC_MrcSetMemoryAccessConfig(TRDC_Type *base, const *trdc_memory_access_control_config_t* *config, *uint8_t* mrcIdx, *uint8_t* regIdx)

Sets the memory access configuration for one of the access control register of one MRC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;

config.securePrivX = true;
config.securePrivW = true;
config.securePrivR = true;
config.lock = true;
TRDC_SetMrcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mrcIdx – MRC index.
- regIdx – Register number.

```
void TRDC_MrcEnableDomainNseUpdate(TRDC_Type *base, uint8_t mrcIdx, uint16_t
                                   domainMask, bool enable)
```

Enables the update of the selected domains.

After the domains' update are enabled, their regions' NSE bits can be set or clear.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains to be enabled.
- enable – True to enable, false to disable.

```
void TRDC_MrcRegionNseSet(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)
```

Sets the NSE bits of the selected regions for domains.

This function sets the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to set.

```
void TRDC_MrcRegionNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)
```

Clears the NSE bits of the selected regions for domains.

This function clears the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to clear.

```
void TRDC_MrcDomainNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t domainMask)
```

Clears the NSE bits for all the regions of the selected domains.

This function clears the NSE bits for all regions of selected domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains whose NSE bits to clear.

```
void TRDC__MrcSetRegionDescriptorConfig(TRDC_Type *base, const
                                         trdc_mrc_region_descriptor_config_t *config)
```

Sets the configuration for one of the region descriptor per domain per MRC instance.

This function sets the configuration for one of the region descriptor, including the start and end address of the region, memory access control policy and valid.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to region descriptor configuration structure.

```
static inline void TRDC__SetMbcGlobalValid(TRDC_Type *base)
```

Sets the TRDC MBC(Memory Block Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

```
void TRDC__GetMbcHardwareConfig(TRDC_Type *base, trdc_slave_memory_hardware_config_t
                                *config, uint8_t mbcIdx, uint8_t slvIdx)
```

Gets the hardware configuration of the one of two slave memories within each MBC(memory block checker).

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the structure to get the configuration.
- mbcIdx – MBC number.
- slvIdx – Slave number.

```
void TRDC__MbcSetNseUpdateConfig(TRDC_Type *base, const trdc_mbc_nse_update_config_t
                                  *config, uint8_t mbcIdx)
```

Sets the NSR update configuration for one of the MBC instance.

After set the NSE configuration, the configured memory area can be updateby NSE set/clear.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to NSE update configuration structure.
- mbcIdx – MBC index.

```
void TRDC__MbcWordNseSet(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)
```

Sets the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to set.

```
void TRDC_MbcWordNseClear(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)
```

Clears the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to clear.

```
void TRDC_MbcNseClearAll(TRDC_Type *base, uint8_t mbcIdx, uint16_t domainMask, uint8_t slaveMask)
```

Clears all configuration words' NSE bits of the selected domain and memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- domainMask – Mask of the domains whose NSE bits to clear, 0b110 means clear domain 1&2.
- slaveMask – Mask of the slaves whose NSE bits to clear, 0x11 means clear all slave 0&1's NSE bits.

```
void TRDC_MbcSetMemoryAccessConfig(TRDC_Type *base, const  
trdc_memory_access_control_config_t *config, uint8_t  
mbcIdx, uint8_t rgdIdx)
```

Sets the memory access configuration for one of the region descriptor of one MBC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;  
  
config.securePrivX = true;  
config.securePrivW = true;  
config.securePrivR = true;  
config.lock = true;  
TRDC_SetMbcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mbcIdx – MBC index.
- rgdIdx – Region descriptor number.

```
void TRDC_MbcSetMemoryBlockConfig(TRDC_Type *base, const  
trdc_mbc_memory_block_config_t *config)
```

Sets the configuration for one of the memory block per domain per MBC instance.

This function sets the configuration for one of the memory block, including the memory access control policy and nse enable.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to memory block configuration structure.

enum _trdc_did_sel

TRDC domain ID select method, the register bit TRDC_MDA_W0_0_DFMT0[DIDS], used for domain hit evaluation.

Values:

enumerator kTRDC_DidMda

Use MDAn[2:0] as DID.

enumerator kTRDC_DidInput

Use the input DID (DID_in) as DID.

enumerator kTRDC_DidMdaAndInput

Use MDAn[2] concatenated with DID_in[1:0] as DID.

enumerator kTRDC_DidReserved

Reserved.

enum _trdc_secure_attr

TRDC secure attribute, the register bit TRDC_MDA_W0_0_DFMT0[SA], used for bus master domain assignment.

Values:

enumerator kTRDC_ForceSecure

Force the bus attribute for this master to secure.

enumerator kTRDC_ForceNonSecure

Force the bus attribute for this master to non-secure.

enumerator kTRDC_MasterSecure

Use the bus master's secure/nonsecure attribute directly.

enumerator kTRDC_MasterSecure1

Use the bus master's secure/nonsecure attribute directly.

enum _trdc_privilege_attr

TRDC privileged attribute, the register bit TRDC_MDA_W0_x_DFMT1[PA], used for non-processor bus master domain assignment.

Values:

enumerator kTRDC_ForceUser

Force the bus attribute for this master to user.

enumerator kTRDC_ForcePrivilege

Force the bus attribute for this master to privileged.

enumerator kTRDC_MasterPrivilege

Use the bus master's attribute directly.

enumerator kTRDC_MasterPrivilege1

Use the bus master's attribute directly.

enum _trdc_controller

TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call TRDC_GetHardwareConfig to get the actual count.

Values:

enumerator kTRDC_MemBlockController0

Memory block checker 0.

enumerator kTRDC_MemBlockController1

Memory block checker 1.

enumerator kTRDC_MemBlockController2

Memory block checker 2.

enumerator kTRDC_MemBlockController3

Memory block checker 3.

enumerator kTRDC_MemRegionChecker0

Memory region checker 0.

enumerator kTRDC_MemRegionChecker1

Memory region checker 1.

enumerator kTRDC_MemRegionChecker2

Memory region checker 2.

enumerator kTRDC_MemRegionChecker3

Memory region checker 3.

enumerator kTRDC_MemRegionChecker4

Memory region checker 4.

enumerator kTRDC_MemRegionChecker5

Memory region checker 5.

enumerator kTRDC_MemRegionChecker6

Memory region checker 6.

enum _trdc_error_state

TRDC domain error state	definition	TRDC_MBCn_DERR_W1[EST]	or
TRDC_MRCn_DERR_W1[EST].			

Values:

enumerator kTRDC_ErrorStateNone

No access violation detected.

enumerator kTRDC_ErrorStateNone1

No access violation detected.

enumerator kTRDC_ErrorStateSingle

Single access violation detected.

enumerator kTRDC_ErrorStateMulti

Multiple access violation detected.

enum _trdc_error_attr

TRDC domain error attribute	definition	TRDC_MBCn_DERR_W1[EATR]	or
TRDC_MRCn_DERR_W1[EATR].			

Values:

enumerator kTRDC_ErrorSecureUserInst

Secure user mode, instruction fetch access.

enumerator kTRDC_ErrorSecureUserData

Secure user mode, data access.

enumerator kTRDC_ErrorSecurePrivilegeInst

Secure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorSecurePrivilegeData

Secure privileged mode, data access.

enumerator kTRDC_ErrorNonSecureUserInst

NonSecure user mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecureUserData

NonSecure user mode, data access.

enumerator kTRDC_ErrorNonSecurePrivilegeInst

NonSecure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecurePrivilegeData

NonSecure privileged mode, data access.

enum _trdc_error_type

TRDC domain error access type definition TRDC_DERR_W1_n[ERW].

Values:

enumerator kTRDC_ErrorTypeRead

Error occurs on read reference.

enumerator kTRDC_ErrorTypeWrite

Error occurs on write reference.

enum _trdc_region_descriptor

The region descriptor enumeration, used to form a mask to set/clear the NSE bits for one or several regions.

Values:

enumerator kTRDC_RegionDescriptor0

Region descriptor 0.

enumerator kTRDC_RegionDescriptor1

Region descriptor 1.

enumerator kTRDC_RegionDescriptor2

Region descriptor 2.

enumerator kTRDC_RegionDescriptor3

Region descriptor 3.

enumerator kTRDC_RegionDescriptor4

Region descriptor 4.

enumerator kTRDC_RegionDescriptor5

Region descriptor 5.

enumerator kTRDC_RegionDescriptor6

Region descriptor 6.

enumerator kTRDC_RegionDescriptor7

Region descriptor 7.

enumerator kTRDC_RegionDescriptor8

Region descriptor 8.

enumerator kTRDC_RegionDescriptor9

Region descriptor 9.

enumerator kTRDC_RegionDescriptor10

Region descriptor 10.

enumerator kTRDC_RegionDescriptor11

Region descriptor 11.

enumerator kTRDC_RegionDescriptor12

Region descriptor 12.

enumerator kTRDC_RegionDescriptor13

Region descriptor 13.

enumerator kTRDC_RegionDescriptor14

Region descriptor 14.

enumerator kTRDC_RegionDescriptor15

Region descriptor 15.

enum _trdc_MRC_domain

The MRC domain enumeration, used to form a mask to enable/disable the update or clear all NSE bits of one or several domains.

Values:

enumerator kTRDC_MrcDomain0

Domain 0.

enumerator kTRDC_MrcDomain1

Domain 1.

enumerator kTRDC_MrcDomain2

Domain 2.

enumerator kTRDC_MrcDomain3

Domain 3.

enumerator kTRDC_MrcDomain4

Domain 4.

enumerator kTRDC_MrcDomain5

Domain 5.

enumerator kTRDC_MrcDomain6

Domain 6.

enumerator kTRDC_MrcDomain7

Domain 7.

enumerator kTRDC_MrcDomain8

Domain 8.

enumerator kTRDC_MrcDomain9

Domain 9.

enumerator kTRDC_MrcDomain10

Domain 10.

enumerator kTRDC_MrcDomain11

Domain 11.

enumerator kTRDC_MrcDomain12

Domain 12.

enumerator kTRDC_MrcDomain13

Domain 13.

enumerator kTRDC_MrcDomain14

Domain 14.

enumerator kTRDC_MrcDomain15

Domain 15.

enum _trdc_MBC_domain

The MBC domain enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several domains.

Values:

enumerator kTRDC_MbcDomain0

Domain 0.

enumerator kTRDC_MbcDomain1

Domain 1.

enumerator kTRDC_MbcDomain2

Domain 2.

enumerator kTRDC_MbcDomain3

Domain 3.

enumerator kTRDC_MbcDomain4

Domain 4.

enumerator kTRDC_MbcDomain5

Domain 5.

enumerator kTRDC_MbcDomain6

Domain 6.

enumerator kTRDC_MbcDomain7

Domain 7.

enum _trdc_MBC_memory

The MBC slave memory enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several memory block.

Values:

enumerator kTRDC_MbcSlaveMemory0

Memory 0.

enumerator kTRDC_MbcSlaveMemory1

Memory 1.

enumerator kTRDC_MbcSlaveMemory2

Memory 2.

enumerator kTRDC_MbcSlaveMemory3

Memory 3.

enum _trdc_MBC_bit

The MBC bit enumeration, used to form a mask to set/clear configured words' NSE.

Values:

enumerator kTRDC_MbcBit0

Bit 0.

enumerator kTRDC_MbcBit1

Bit 1.

enumerator kTRDC_MbcBit2
Bit 2.

enumerator kTRDC_MbcBit3
Bit 3.

enumerator kTRDC_MbcBit4
Bit 4.

enumerator kTRDC_MbcBit5
Bit 5.

enumerator kTRDC_MbcBit6
Bit 6.

enumerator kTRDC_MbcBit7
Bit 7.

enumerator kTRDC_MbcBit8
Bit 8.

enumerator kTRDC_MbcBit9
Bit 9.

enumerator kTRDC_MbcBit10
Bit 10.

enumerator kTRDC_MbcBit11
Bit 11.

enumerator kTRDC_MbcBit12
Bit 12.

enumerator kTRDC_MbcBit13
Bit 13.

enumerator kTRDC_MbcBit14
Bit 14.

enumerator kTRDC_MbcBit15
Bit 15.

enumerator kTRDC_MbcBit16
Bit 16.

enumerator kTRDC_MbcBit17
Bit 17.

enumerator kTRDC_MbcBit18
Bit 18.

enumerator kTRDC_MbcBit19
Bit 19.

enumerator kTRDC_MbcBit20
Bit 20.

enumerator kTRDC_MbcBit21
Bit 21.

enumerator kTRDC_MbcBit22
Bit 22.

enumerator `kTRDC_MbcBit23`

Bit 23.

enumerator `kTRDC_MbcBit24`

Bit 24.

enumerator `kTRDC_MbcBit25`

Bit 25.

enumerator `kTRDC_MbcBit26`

Bit 26.

enumerator `kTRDC_MbcBit27`

Bit 27.

enumerator `kTRDC_MbcBit28`

Bit 28.

enumerator `kTRDC_MbcBit29`

Bit 29.

enumerator `kTRDC_MbcBit30`

Bit 30.

enumerator `kTRDC_MbcBit31`

Bit 31.

typedef struct *_trdc_hardware_config* trdc_hardware_config_t

TRDC hardware configuration.

typedef struct *_trdc_slave_memory_hardware_config* trdc_slave_memory_hardware_config_t

Hardware configuration of the two slave memories within each MBC(memory block checker).

typedef enum *_trdc_did_sel* trdc_did_sel_t

TRDC domain ID select method, the register bit `TRDC_MDA_W0_0_DFMT0[DIDS]`, used for domain hit evaluation.

typedef enum *_trdc_secure_attr* trdc_secure_attr_t

TRDC secure attribute, the register bit `TRDC_MDA_W0_0_DFMT0[SA]`, used for bus master domain assignment.

typedef struct *_trdc_processor_domain_assignment* trdc_processor_domain_assignment_t

Domain assignment for the processor bus master.

typedef enum *_trdc_privilege_attr* trdc_privilege_attr_t

TRDC privileged attribute, the register bit `TRDC_MDA_W0_x_DFMT1[PA]`, used for non-processor bus master domain assignment.

typedef struct *_trdc_non_processor_domain_assignment*

trdc_non_processor_domain_assignment_t

Domain assignment for the non-processor bus master.

typedef struct *_trdc_idau_config* trdc_idau_config_t

IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

typedef struct *_trdc_flw_config* trdc_flw_config_t

FLW(Flash Logical Window) configuration.

typedef enum *_trdc_controller* trdc_controller_t

TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call `TRDC_GetHardwareConfig` to get the actual count.

typedef enum *_trdc_error_state* trdc_error_state_t
TRDC domain error state definition TRDC_MBCn_DERR_W1[EST] or
TRDC_MRCn_DERR_W1[EST].

typedef enum *_trdc_error_attr* trdc_error_attr_t
TRDC domain error attribute definition TRDC_MBCn_DERR_W1[EATR] or
TRDC_MRCn_DERR_W1[EATR].

typedef enum *_trdc_error_type* trdc_error_type_t
TRDC domain error access type definition TRDC_DERR_W1_n[ERW].

typedef struct *_trdc_domain_error* trdc_domain_error_t
TRDC domain error definition.

typedef struct *_trdc_memory_access_control_config* trdc_memory_access_control_config_t
Memory access control configuration for MBC/MRC.

typedef struct *_trdc_mrc_region_descriptor_config* trdc_mrc_region_descriptor_config_t
The configuration of each region descriptor per domain per MRC instance.

typedef struct *_trdc_mbc_nse_update_config* trdc_mbc_nse_update_config_t
The configuration of MBC NSE update.

typedef struct *_trdc_mbc_memory_block_config* trdc_mbc_memory_block_config_t
The configuration of each memory block per domain per MBC instance.

FSL_TRDC_DRIVER_VERSION

struct *_trdc_hardware_config*
#include <fsl_trdc.h> TRDC hardware configuration.

Public Members

uint8_t masterNumber
Number of bus masters.

uint8_t domainNumber
Number of domains.

uint8_t mbcNumber
Number of MBCs.

uint8_t mrcNumber
Number of MRCs.

struct *_trdc_slave_memory_hardware_config*
#include <fsl_trdc.h> Hardware configuration of the two slave memories within each
MBC(memory block checker).

Public Members

uint32_t blockNum
Number of blocks.

uint32_t blockSize
Block size.

struct *_trdc_processor_domain_assignment*
#include <fsl_trdc.h> Domain assignment for the processor bus master.

Public Members

uint32_t domainId

Domain ID.

uint32_t domainIdSelect

Domain ID select method, see trdc_did_sel_t.

uint32_t __pad0__

Reserved.

uint32_t secureAttr

Secure attribute, see trdc_secure_attr_t.

uint32_t __pad1__

Reserved.

uint32_t lock

Lock the register.

uint32_t __pad2__

Reserved.

struct __trdc_non_processor_domain_assignment

#include <fsl_trdc.h> Domain assignment for the non-processor bus master.

Public Members

uint32_t domainId

Domain ID.

uint32_t privilegeAttr

Privileged attribute, see trdc_privilege_attr_t.

uint32_t secureAttr

Secure attribute, see trdc_secure_attr_t.

uint32_t bypassDomainId

Bypass domain ID.

uint32_t __pad0__

Reserved.

uint32_t lock

Lock the register.

uint32_t __pad1__

Reserved.

struct __trdc_idau_config

#include <fsl_trdc.h> IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

Public Members

uint32_t __pad0__

Reserved.

uint32_t lockSecureVTOR

Disable writes to secure VTOR(Vector Table Offset Register).

uint32_t lockNonsecureVTOR

Disable writes to non-secure VTOR, Application interrupt and Reset Control Registers.

uint32_t lockSecureMPU

Disable writes to secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor in Secure state.

uint32_t lockNonsecureMPU

Disable writes to non-secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor.

uint32_t lockSAU

Disable writes to SAU(Security Attribution Unit) registers.

uint32_t __pad1__

Reserved.

struct __trdc_flw_config

#include <fsl_trdc.h> FLW(Flash Logical Window) configuration.

Public Members

uint16_t blockCount

Block count of the Flash Logic Window in 32KByte blocks.

uint32_t arrayBaseAddr

Flash array base address of the Flash Logical Window.

bool lock

Disable writes to FLW registers.

bool enable

Enable FLW function.

struct __trdc_domain_error

#include <fsl_trdc.h> TRDC domain error definition.

Public Members

trdc_controller_t controller

Which controller captured access violation.

uint32_t address

Access address that generated access violation.

trdc_error_state_t errorState

Error state.

trdc_error_attr_t errorAttr

Error attribute.

trdc_error_type_t errorType

Error type.

uint8_t errorPort

Error port.

uint8_t domainId

Domain ID.

uint8_t slaveMemoryIdx

The slave memory index. Only apply when violation in MBC.

struct __trdc_memory_access_control_config

#include <fsl_trdc.h> Memory access control configuration for MBC/MRC.

Public Members

uint32_t nonsecureUsrX

Allow nonsecure user execute access.

uint32_t nonsecureUsrW

Allow nonsecure user write access.

uint32_t nonsecureUsrR

Allow nonsecure user read access.

uint32_t __pad0__

Reserved.

uint32_t nonsecurePrivX

Allow nonsecure privilege execute access.

uint32_t nonsecurePrivW

Allow nonsecure privilege write access.

uint32_t nonsecurePrivR

Allow nonsecure privilege read access.

uint32_t __pad1__

Reserved.

uint32_t secureUsrX

Allow secure user execute access.

uint32_t secureUsrW

Allow secure user write access.

uint32_t secureUsrR

Allow secure user read access.

uint32_t __pad2__

Reserved.

uint32_t securePrivX

Allownonsecure privilege execute access.

uint32_t securePrivW

Allownonsecure privilege write access.

uint32_t securePrivR

Allownonsecure privilege read access.

uint32_t __pad3__

Reserved.

uint32_t lock

Lock the configuration until next reset, only apply to access control register 0.

struct __trdc_mrc_region_descriptor_config

#include <fsl_trdc.h> The configuration of each region descriptor per domain per MRC instance.

Public Members

uint8_t memoryAccessControlSelect

Select one of the 8 access control policies for this region, for access control policies see `trdc_memory_access_control_config_t`.

uint32_t startAddr

Physical start address.

bool valid

Lock the register.

bool nseEnable

Enable non-secure accesses and disable secure accesses.

uint32_t endAddr

Physical start address.

uint8_t mrcIdx

The index of the MRC for this configuration to take effect.

uint8_t domainIdx

The index of the domain for this configuration to take effect.

uint8_t regionIdx

The index of the region for this configuration to take effect.

struct `_trdc_mbc_nse_update_config`

#include <fsl_trdc.h> The configuration of MBC NSE update.

Public Members

uint32_t __pad0__

Reserved.

uint32_t wordIdx

MBC configuration word index to be updated.

uint32_t __pad1__

Reserved.

uint32_t memorySelect

Bit mask of the selected memory to be updated. `_trdc_MBC_memory`.

uint32_t __pad2__

Reserved.

uint32_t domainSelect

Bit mask of the selected domain to be updated. `_trdc_MBC_domain`.

uint32_t __pad3__

Reserved.

uint32_t autoIncrement

Whether to increment the word index after current word is updated using this configuration.

struct `_trdc_mbc_memory_block_config`

#include <fsl_trdc.h> The configuration of each memory block per domain per MBC instance.

Public Members

`uint32_t memoryAccessControlSelect`

Select one of the 8 access control policies for this memory block, for access control policies see `trdc_memory_access_control_config_t`.

`uint32_t nseEnable`

Enable non-secure accesses and disable secure accesses.

`uint32_t mbcIdx`

The index of the MBC for this configuration to take effect.

`uint32_t domainIdx`

The index of the domain for this configuration to take effect.

`uint32_t slaveMemoryIdx`

The index of the slave memory for this configuration to take effect.

`uint32_t memoryBlockIdx`

The index of the memory block for this configuration to take effect.

2.78 TRGMUX: Trigger Mux Driver

`static inline void TRGMUX_LockRegister(TRGMUX_Type *base, uint32_t index)`

Sets the flag of the register which is used to mark writeable.

The function sets the flag of the register which is used to mark writeable. Example:

```
TRGMUX_LockRegister(TRGMUX0, kTRGMUX_Trgmux0Dmamux0);
```

Parameters

- `base` – TRGMUX peripheral base address.
- `index` – The index of the TRGMUX register, see the enum `trgmux_device_t` defined in `<SOC>.h`.

`status_t TRGMUX_SetTriggerSource(TRGMUX_Type *base, uint32_t index, trgmux_trigger_input_t input, uint32_t trigger_src)`

Configures the trigger source of the appointed peripheral.

The function configures the trigger source of the appointed peripheral. Example:

```
TRGMUX_SetTriggerSource(TRGMUX0, kTRGMUX_Trgmux0Dmamux0, kTRGMUX_TriggerInput0,
↪ kTRGMUX_SourcePortPin);
```

Parameters

- `base` – TRGMUX peripheral base address.
- `index` – The index of the TRGMUX register, see the enum `trgmux_device_t` defined in `<SOC>.h`.
- `input` – The MUX select for peripheral trigger input
- `trigger_src` – The trigger inputs for various peripherals. See the enum `trgmux_source_t` defined in `<SOC>.h`.

Return values

- `kStatus_Success` – Configured successfully.

- `kStatus_TRGMUX_Locked` – Configuration failed because the register is locked.

`FSL_TRGMUX_DRIVER_VERSION`

TRGMUX driver version.

TRGMUX configure status.

Values:

enumerator `kStatus_TRGMUX_Locked`

Configure failed for register is locked

enum `_trgmux_trigger_input`

Defines the MUX select for peripheral trigger input.

Values:

enumerator `kTRGMUX_TriggerInput0`

The MUX select for peripheral trigger input 0

enumerator `kTRGMUX_TriggerInput1`

The MUX select for peripheral trigger input 1

enumerator `kTRGMUX_TriggerInput2`

The MUX select for peripheral trigger input 2

enumerator `kTRGMUX_TriggerInput3`

The MUX select for peripheral trigger input 3

typedef enum `_trgmux_trigger_input` `trgmux_trigger_input_t`

Defines the MUX select for peripheral trigger input.

2.79 TSTMR: Timestamp Timer Driver

`FSL_TSTMR_DRIVER_VERSION`

Version 2.0.3

static inline uint64_t `TSTMR_ReadTimeStamp(TSTMR_Type *base)`

Reads the time stamp.

This function reads the low and high registers and returns the 56-bit free running counter value. This can be read by software at any time to determine the software ticks. TSTMR registers can be read with 32-bit accesses only. The TSTMR LOW read should occur first, followed by the TSTMR HIGH read.

Parameters

- `base` – TSTMR peripheral base address.

Returns

The 56-bit time stamp value.

static inline void `TSTMR_DelayUs(TSTMR_Type *base, uint64_t delayInUs)`

Delays for a specified number of microseconds.

This function repeatedly reads the timestamp register and waits for the user-specified delay value.

Parameters

- `base` – TSTMR peripheral base address.

- delayInUs – Delay value in microseconds.

FSL_COMPONENT_ID

2.80 Upower

enum drive_mode_e

Drive Mode: OD(Over Drive), ND(Nominal Drive), UD(Under Drive)

Values:

enumerator DRIVE_MODE_UD

enumerator DRIVE_MODE_ND

enumerator DRIVE_MODE_OD

enumerator DRIVE_MODE_NUM

enum _upower_ps_mask

upower power switch mask definition.

Values:

enumerator kUPOWER_PS_M33

RTD: CM33 core complex/platform/peripherals

enumerator kUPOWER_PS_FUSION

RTD: Fusion core and peripherals

enumerator kUPOWER_PS_A35_0

AD: A35[0] core complex

enumerator kUPOWER_PS_A35_1

AD: A35[1] core complex

enumerator kUPOWER_PS_L2_CACHE

AD: 256KB L2 cache

enumerator kUPOWER_PS_AD_NIC

AD: A35 NIC_PER, NIC_AP

enumerator kUPOWER_PS_AD_PER

AD: A35 peripherals and croessbar

enumerator kUPOWER_PS_GPU3D

LPAV: 3DGPU

enumerator kUPOWER_PS_HIFI4

LPAV: HiFi4 complex

enumerator kUPOWER_PS_DDRC

LPAV: DDR controller

enumerator kUPOWER_PS_PXP_EPDC

LPAV: PXP, EPDC

enumerator kUPOWER_PS_MIPI_DSI

LPAV: MIPI DSI

enumerator kUPOWER_PS_MIPI_CSI

LPAV: MIPI CSI

enumerator kUPOWER_PS_AV_NIC

LPAV: NIC, System RAM2, DCNano

enumerator kUPOWER_PS_FUSION_AO

RTD: Fusion “Always ON”

enumerator kUPOWER_PS_FUSES

RTD: Fuses

enum __upower_mp0_mask

upower memory partitions 0 mask definition.

Values:

enumerator kUPOWER_MP0_A35_0

AD: A35 core0

enumerator kUPOWER_MP0_A35_1

AD: A35 core1

enumerator kUPOWER_MP0_A35_CORE_A

AD: A35 core0/1

enumerator kUPOWER_MP0_A35_CORE_B

AD: A35 core0/1

enumerator kUPOWER_MP0_A35_CORE_C

AD: A35 core0/1

enumerator kUPOWER_MP0_CAAM

AD: CAAM

enumerator kUPOWER_MP0_DMA1

AD: DMA1

enumerator kUPOWER_MP0_FLEXSPI2

AD: FlexSPI2

enumerator kUPOWER_MP0_AD_SYSTEM_A

AD: System

enumerator kUPOWER_MP0_AD_SYSTEM_B

AD: System

enumerator kUPOWER_MP0_USB

AD: USB

enumerator kUPOWER_MP0_USDHC0

AD: uSDHC0

enumerator kUPOWER_MP0_USDHC1

AD: uSDHC1

enumerator kUPOWER_MP0_USDHC2

AD: uSDHC2

enumerator kUPOWER_MP0_GIC

AD: GIC

enumerator kUPOWER_MP0_ENET

AD: ENET

enumerator kUPOWER_MP0_BRAINSHIFT

AD: Brainshift

enumerator kUPOWER_MP0_DCNANO_A

LPAV: DCNano

enumerator kUPOWER_MP0_DCNANO_B

LPAV: DCNano

enumerator kUPOWER_MP0_EPDC_A

LPAV: EPDC

enumerator kUPOWER_MP0_EPDC_B

LPAV: EPDC

enumerator kUPOWER_MP0_DMA2

LPAV: DMA2

enumerator kUPOWER_MP0_GPU2D_A

LPAV: GPU2D

enumerator kUPOWER_MP0_GPU2D_B

LPAV: GPU2D

enumerator kUPOWER_MP0_GPU3D_A

LPAV: GPU3D

enumerator kUPOWER_MP0_GPU3D_B

LPAV: GPU3D

enumerator kUPOWER_MP0_HIFI4

LPAV: Hi-Fi DSP

enumerator kUPOWER_MP0_ISI

LPAV: ISI

enumerator kUPOWER_MP0_MIPI_CSI

LPAV: MIPI CSI

enumerator kUPOWER_MP0_MIPI_DSI

LPAV: MIPI DSI

enumerator kUPOWER_MP0_PXP

LPAV: PXP

enumerator kUPOWER_MP0_AV_SYSTEM

LPAV: System

enum _upower_mp1_mask

upower memory partitions 1 mask definition.

Values:

enumerator kUPOWER_MP1_CASPER

RTD: Casper

enumerator kUPOWER_MP1_DMA0

RTD: DMA0

enumerator kUPOWER_MP1_FLEXCAN

RTD: FlexCAN (CAN-FD)

enumerator kUPOWER_MP1_FLEXSPI0
RTD: FlexSPI0 (OTFAD)

enumerator kUPOWER_MP1_FLEXSPI1
RTD: FlexSPI1

enumerator kUPOWER_MP1_M33
RTD: M33

enumerator kUPOWER_MP1_POWERQUAD
RTD: PowerQuad

enumerator kUPOWER_MP1_M33 ETF
RTD: M33-ETF

enumerator kUPOWER_MP1_SENTINEL_A
RTD: Sentinel

enumerator kUPOWER_MP1_SENTINEL_B
RTD: Sentinel

enumerator kUPOWER_MP1_UPOWER_A
RTD: uPower

enumerator kUPOWER_MP1_UPOWER_B
RTD: uPower

enumerator kUPOWER_MP1_RTD_SYSTEM
RTD: System

enumerator kUPOWER_MP1_SYSTEM0
RTD: System[0]

enumerator kUPOWER_MP1_SYSTEM1
RTD: System[1]

enumerator kUPOWER_MP1_SYSTEM2
RTD: System[2]

enumerator kUPOWER_MP1_SYSTEM3
RTD: System[3]

enumerator kUPOWER_MP1_SYSTEM4
RTD: System[4]

enumerator kUPOWER_MP1_SYSTEM5
RTD: System[5]

enumerator kUPOWER_MP1_SYSTEM6
RTD: System[6]

enumerator kUPOWER_MP1_SYSTEM7
RTD: System[7]

enumerator kUPOWER_MP1_SENTINEL_C
RTD: Sentinel

typedef enum *_upower_ps_mask* upower_ps_mask_t
upower power switch mask definition.

typedef enum *_upower_mp0_mask* upower_mp0_mask_t
upower memory partitions 0 mask definition.

```
typedef enum _upower_mp1_mask upower_mp1_mask_t
    upower memory partitions 1 mask definition.
```

```
typedef struct _upower_version upower_version_t
```

```
void UPOWER_CheckReqWithArgs(upwr_sg_t sg, uint32_t *sgfptr, upwr_resp_t *errptr, int
    *retptr, uint32_t attempts)
```

Check status of current request to uPower.

Will block till related callback function has been called

@sg Indicate which Service Group that request point to @sgfptr: pointer to the variable that will hold the function id of the last request completed @errptr: pointer to the variable that will hold the error code @retptr: pointer to the variable that will hold the value returned by the last request completed (invalid if the last request completed didn't return any value) @attempts: maximum number of polling attempts; if attempts > 0 and is reached with no service response received, upwr_poll_req_status returns UPWR_REQ_BUSY and variables pointed by sgfptr, retptr and errptr are not updated; if attempts = 0, upwr_poll_req_status waits "forever".

```
void uPower_IRQHandler(void)
    uPower specific IRQ handler.
```

```
void UPOWER_CheckReq(upwr_sg_t sg)
    Check status of current request to uPower.
    Will block till related callback function has been called
    @sg Indicate which Service Group that request point to
```

```
void UPOWER_Init(upower_version_t *pVersion)
    Initialize MU interface for uPower access.
```

Parameters

- pVersion – Pointer to the structure to save uPower ROM and RAM versions

```
void UPOWER_Deinit(void)
    Deinitialize MU interface for uPower access.
```

```
int UPOWER_PowerOnDomain(soc_domain_t domain)
    Power on certain domain without reset.
```

Parameters

- domain – Target domain to power on.

Returns

0 if ok, failure otherwise.

```
int UPOWER_BootDomain(soc_domain_t domain)
    Reset and kick off certain domain.
```

Parameters

- domain – Target domain to boot.

Returns

0 if ok, failure otherwise.

```
int UPOWER_PowerOnSwitches(upower_ps_mask_t mask)
    Power on power switches.
```

Parameters

- mask – Bits to define which switch should be turned on. The value should be ORed by upower_ps_mask_t. Bit value 1 means the switch is to be powered on; Value 0 means the switch keeps unchanged.

Returns

0 if ok, failure otherwise.

int UPOWER_PowerOffSwitches(*upower_ps_mask_t* mask)

Power off power switches.

Parameters

- mask – Bits to define which switch should be turned off. The value should be ORed by *upower_ps_mask_t*. Bit value 1 means the switch is to be powered off; Value 0 means the switch keeps unchanged.

Returns

0 if ok, failure otherwise.

int UPOWER_PowerOnMemPart(uint32_t mask0, uint32_t mask1)

Power on memory partitions array/periphery logic. If a memory is requested to turn on, but the power switch that feeds that memory is not, the power switch will be turned on automatically.

The parameter mask bits define which memory partition should be turned on. The value should be ORed by *upower_mp0_mask_t* or *upower_mp1_mask_t*. Mask bit value 1 means the switch is to be powered on; Value 0 means the switch keeps unchanged.

Parameters

- mask0 – memory partition group 0 mask, see *upower_mp0_mask_t*.
- mask1 – memory partition group 1 mask, see *upower_mp1_mask_t*.

Returns

0 if ok, failure otherwise.

int UPOWER_PowerOffMemPart(uint32_t mask0, uint32_t mask1)

Power off memory partitions array/periphery logic.

The parameter mask bits define which memory partition should be turned off. The value should be ORed by *upower_mp0_mask_t* and *upower_mp1_mask_t*. Mask bit value 1 means the switch is to be powered off; Value 0 means the switch keeps unchanged.

Parameters

- mask0 – memory partition group 0 mask, see *upower_mp0_mask_t*.
- mask1 – memory partition group 1 mask, see *upower_mp1_mask_t*.

Returns

0 if ok, failure otherwise.

int UPOWER_RetainMemPart(uint32_t mask0, uint32_t mask1)

Power on memory partitions array logic and power off its periphery logic. If a memory array is requested to turn on, but the power switch that feeds that memory is not, the power switch will be turned on automatically.

The parameter mask bits define which memory partition should be turned on. The value should be ORed by *upower_mp0_mask_t* or *upower_mp1_mask_t*. Mask bit value 1 means the switch is to be powered on; Value 0 means the switch keeps unchanged.

Parameters

- mask0 – memory partition group 0 mask, see *upower_mp0_mask_t*.
- mask1 – memory partition group 1 mask, see *upower_mp1_mask_t*.

Returns

0 if ok, failure otherwise.

int UPOWER_SetRtdUseDdr(bool use_ddr)

M33 call this API to inform uPower, M33 is using ddr.

Parameters

- use_ddr – not 0, true, means that RTD is using ddr. 0, false, means that, RTD is not using ddr.

Returns

0 if ok, failure otherwise(-1 if service group is busy, -3 if called in an invalid API state).

int UPOWER_PowerOnADInPDMMode(void)

M33 call this API to Power On Application Domain when Application Domain is in Power Down/Deep Power Down mode. After upower get the msg, upower write a flag to registers according to msg. Then upower will get a WUU1 interrupt request and upower change Power Down mode/Deep Power mode to Active mode when Application Domain is in Power Down/Deep Power Down mode.

Returns

0 if ok, failure otherwise.

int UPOWER_SetPwrMgmtParam(upwr_pwm_param_t *pcfg_in)

set sg powermanagement parameters of upower.

please see upower_soc_defs.h file for struct upwr_pwm_param_t

Parameters

- pcfg_in – user's selection for powermanagement parameter of upower

Returns

0 if ok, failure otherwise.

int UPOWER_ReduceBuck23VoltInSTBY(void)

Reduce Buck2 and Buck3 voltage in STANDBY mode.

Returns

0 if ok, failure otherwise.

int UPOWER_SetDDRRetention(soc_domain_t domain, bool enable)

Set DDR retention control bit.

Parameters

- domain – Target domain to power on.
- enable – true means set ddr retention, false means clear ddr retention.

Returns

0 if ok, failure otherwise.

int UPOWER_SetLDO1Power(bool enable)

Set LDO1 rail power.

Parameters

- enable – True means on, false means off.

Returns

0 if ok, failure otherwise.

int UPOWER_SetBuck3Power(bool enable)

Set Buck3 rail power.

Parameters

- enable – True means on, false means off.

Returns

0 if ok, failure otherwise.

int UPOWER__ChngPmicVoltage(uint32_t rail, int voltage)

int UPOWER__SetPmicReg(uint32_t reg_addr, uint32_t reg_val)

Write PMIC registers by I2C.

Parameters

- reg_addr – means pmic ic target address.
- reg_val – means write data, specific data can be found in PCA9460.

Returns

0 if ok, failure otherwise.

int UPOWER__ReadPmicReg(uint32_t reg_addr, uint8_t *reg_val)

Read PMIC registers by I2C.

Parameters

- reg_addr – means pmic ic pca9460 target address.
- reg_val – means value of register.

Returns

0 if ok, failure otherwise.

void UPOWER__InitBuck2Buck3Table(void)

Init BUCK2 BUCK3 output voltage table.

int UPOWER__GetPmicVoltage(uint32_t rail, int *voltage)

get PMIC voltage.

Returns

0 if ok, failure otherwise.

int UPOWER__ChngRTDDomBias(*drive_mode_e* drive_mode)

Change Bias Configuration(biasing options: NBB, RBB, ARBB, AFBB; Bias Voltage) for RTD.

Parameters

- drive_mode – Drive Mode: UD, ND, OD.

Returns

0 if ok, failure otherwise.

FSL_UPOWER__DRIVER_VERSION

upower driver version (2.0.19.)

PMIC__BUCK1

pmic rail id.

PMIC__BUCK2

PMIC__BUCK3

PMIC__BUCK4

PMIC__LDO1

PMIC__LDO2

PMIC__LDO3

PMIC__LDO4

PMIC_LSW1
PMIC_LSW2
PMIC_LSW3
PMIC_LSW4
PMIC_VOLTAGE_1_1V
 pmic voltage.
PMIC_VOLTAGE_1_0V
PMIC_VOLTAGE_0_9V
PMIC_I2C_ADDR
PMIC_WRITE_DATA_SIZE
PMIC_IC_WRITE_DATA_SIZE
PMIC_READ_DATA_SIZE
PMIC_IC_READ_DATA_SIZE
PCA9460_BUCK23_DVS_CFG1_ADDR
 PCA9460 register.
PCA9460_BUCK23_DVS_CFG2_ADDR
PCA9460_BUCK2CTRL_ADDR
PCA9460_BUCK2OUT_DVS0_ADDR
PCA9460_BUCK2OUT_DVS1_ADDR
PCA9460_BUCK2OUT_DVS2_ADDR
PCA9460_BUCK2OUT_DVS3_ADDR
PCA9460_BUCK2OUT_DVS4_ADDR
PCA9460_BUCK2OUT_DVS5_ADDR
PCA9460_BUCK2OUT_DVS6_ADDR
PCA9460_BUCK2OUT_DVS7_ADDR
PCA9460_BUCK2OUT_STDBY_ADDR
PCA9460_BUCK2OUT_MAX_LIMIT_ADDR
PCA9460_BUCK2OUT_MIN_LIMIT_ADDR
PCA9460_BUCK3CTRL_ADDR
PCA9460_BUCK3OUT_DVS0_ADDR
PCA9460_LDO1_CFG_ADDR
PCA9460_LDO1_OUT_ADDR
BUCK23_DVS_CFG2_B2_DVS_CTRL_BITS
 PCA9460 BUCK23_DVS_CFG2 register

BUCK23_DVS_CFG2_B3_DVS_CTRL_BITS

BUCK23_DVS_CFG2_RSVD_BITS

B3_RAMP_BITS

PCA9460 BUCK3CTRL register

B3_LPMODE_BITS

B3_AD_BITS

B3_FPWM_BITS

B3_ENMODE_BITS

BUCK_OUTPUT_TABLE_SIZE

BUCK2_BUCK3_VOLTAGE_STEPS

L1_CSEL_BITS

PCA9460 LDO1_CFG register

L1_LLSEL_BITS

L1_LPMODE_BITS

L1_ENMODE_BITS

uint8_t val

uint8_t RSVD

uint8_t B3_DVS_CTRL

uint8_t B2_DVS_CTRL

struct *pca9460_buck23_dvs_cfg2_t* reg

uint8_t val

uint8_t B3_ENMODE

uint8_t B3_FPWM

uint8_t B3_AD

uint8_t B3_LPMODE

uint8_t B3_RAMP

struct *pca9460_buck3ctrl_t* reg

uint8_t val

uint8_t L1_ENMODE

uint8_t L1_LPMODE

uint8_t L1_LLSEL

uint8_t L1_CSEL

struct *pca9460_ldo1_cfg_t* reg

uint32_t romMajor

```

uint32_t romMinor
uint32_t romFixes
uint32_t ramMajor
uint32_t ramMinor
uint32_t ramFixes

union pca9460_buck23_dvs_cfg2_t
    #include <fsl_upower.h>
union pca9460_buck3ctrl_t
    #include <fsl_upower.h>
union pca9460_ldo1_cfg_t
    #include <fsl_upower.h>
struct __upower__version
    #include <fsl_upower.h>
struct reg
struct reg
struct reg

```

2.81 USDHC: Ultra Secured Digital Host Controller Driver

`void USDHC_Init(USDHC_Type *base, const usdhc_config_t *config)`

USDHC module initialization function.

Configures the USDHC according to the user configuration.

Example:

```

usdhc_config_t config;
config.cardDetectDat3 = false;
config.endianMode = kUSDHC_EndianModeLittle;
config.dmaMode = kUSDHC_DmaModeAdma2;
config.readWatermarkLevel = 128U;
config.writeWatermarkLevel = 128U;
USDHC_Init(USDHC, &config);

```

Parameters

- `base` – USDHC peripheral base address.
- `config` – USDHC configuration information.

Return values

`kStatus_Success` – Operate successfully.

`void USDHC_Deinit(USDHC_Type *base)`

Deinitializes the USDHC.

Parameters

- `base` – USDHC peripheral base address.

`bool USDHC_Reset(USDHC_Type *base, uint32_t mask, uint32_t timeout)`

Resets the USDHC.

Parameters

- `base` – USDHC peripheral base address.
- `mask` – The reset type mask(`_usdhc_reset`).
- `timeout` – Timeout for reset.

Return values

- `true` – Reset successfully.
- `false` – Reset failed.

`status_t USDHC_SetAdmaTableConfig(USDHC_Type *base, usdhc_adma_config_t *dmaConfig, usdhc_data_t *dataConfig, uint32_t flags)`

Sets the DMA descriptor table configuration. A high level DMA descriptor configuration function.

Parameters

- `base` – USDHC peripheral base address.
- `dmaConfig` – ADMA configuration
- `dataConfig` – Data descriptor
- `flags` – ADAM descriptor flag, used to indicate to create multiple or single descriptor, please refer to `enum _usdhc_adma_flag`.

Return values

- `kStatus_OutOfRange` – ADMA descriptor table length isn't enough to describe data.
- `kStatus_Success` – Operate successfully.

`status_t USDHC_SetInternalDmaConfig(USDHC_Type *base, usdhc_adma_config_t *dmaConfig, const uint32_t *dataAddr, bool enAutoCmd23)`

Internal DMA configuration. This function is used to config the USDHC DMA related registers.

Parameters

- `base` – USDHC peripheral base address.
- `dmaConfig` – ADMA configuration.
- `dataAddr` – Transfer data address, a simple DMA parameter, if ADMA is used, leave it to `NULL`.
- `enAutoCmd23` – Flag to indicate Auto CMD23 is enable or not, a simple DMA parameter, if ADMA is used, leave it to `false`.

Return values

- `kStatus_OutOfRange` – ADMA descriptor table length isn't enough to describe data.
- `kStatus_Success` – Operate successfully.

`status_t USDHC_SetADMA2Descriptor(uint32_t *admaTable, uint32_t admaTableWords, const uint32_t *dataBufferAddr, uint32_t dataBytes, uint32_t flags)`

Sets the ADMA2 descriptor table configuration.

Parameters

- `admaTable` – ADMA table address.
- `admaTableWords` – ADMA table length.
- `dataBufferAddr` – Data buffer address.
- `dataBytes` – Data Data length.
- `flags` – ADAM descriptor flag, used to indicate to create multiple or single descriptor, please refer to `enum _usdhc_adma_flag`.

Return values

- `kStatus_OutOfRange` – ADMA descriptor table length isn't enough to describe data.
- `kStatus_Success` – Operate successfully.

```
status_t USDHC_SetADMA1Descriptor(uint32_t *admaTable, uint32_t admaTableWords, const
                                uint32_t *dataBufferAddr, uint32_t dataBytes, uint32_t
                                flags)
```

Sets the ADMA1 descriptor table configuration.

Parameters

- `admaTable` – ADMA table address.
- `admaTableWords` – ADMA table length.
- `dataBufferAddr` – Data buffer address.
- `dataBytes` – Data length.
- `flags` – ADAM descriptor flag, used to indicate to create multiple or single descriptor, please refer to `enum _usdhc_adma_flag`.

Return values

- `kStatus_OutOfRange` – ADMA descriptor table length isn't enough to describe data.
- `kStatus_Success` – Operate successfully.

```
static inline void USDHC_EnableInternalDMA(USDHC_Type *base, bool enable)
```

Enables internal DMA.

Parameters

- `base` – USDHC peripheral base address.
- `enable` – enable or disable flag

```
static inline void USDHC_EnableInterruptStatus(USDHC_Type *base, uint32_t mask)
```

Enables the interrupt status.

Parameters

- `base` – USDHC peripheral base address.
- `mask` – Interrupt status flags mask(`_usdhc_interrupt_status_flag`).

```
static inline void USDHC_DisableInterruptStatus(USDHC_Type *base, uint32_t mask)
```

Disables the interrupt status.

Parameters

- `base` – USDHC peripheral base address.
- `mask` – The interrupt status flags mask(`_usdhc_interrupt_status_flag`).

static inline void USDHC_EnableInterruptSignal(USDHC_Type *base, uint32_t mask)

Enables the interrupt signal corresponding to the interrupt status flag.

Parameters

- base – USDHC peripheral base address.
- mask – The interrupt status flags mask(_usdhc_interrupt_status_flag).

static inline void USDHC_DisableInterruptSignal(USDHC_Type *base, uint32_t mask)

Disables the interrupt signal corresponding to the interrupt status flag.

Parameters

- base – USDHC peripheral base address.
- mask – The interrupt status flags mask(_usdhc_interrupt_status_flag).

static inline uint32_t USDHC_GetEnabledInterruptStatusFlags(USDHC_Type *base)

Gets the enabled interrupt status.

Parameters

- base – USDHC peripheral base address.

Returns

Current interrupt status flags mask(_usdhc_interrupt_status_flag).

static inline uint32_t USDHC_GetInterruptStatusFlags(USDHC_Type *base)

Gets the current interrupt status.

Parameters

- base – USDHC peripheral base address.

Returns

Current interrupt status flags mask(_usdhc_interrupt_status_flag).

static inline void USDHC_ClearInterruptStatusFlags(USDHC_Type *base, uint32_t mask)

Clears a specified interrupt status. write 1 clears.

Parameters

- base – USDHC peripheral base address.
- mask – The interrupt status flags mask(_usdhc_interrupt_status_flag).

static inline uint32_t USDHC_GetAutoCommand12ErrorStatusFlags(USDHC_Type *base)

Gets the status of auto command 12 error.

Parameters

- base – USDHC peripheral base address.

Returns

Auto command 12 error status flags mask(_usdhc_auto_command12_error_status_flag).

static inline uint32_t USDHC_GetAdmaErrorStatusFlags(USDHC_Type *base)

Gets the status of the ADMA error.

Parameters

- base – USDHC peripheral base address.

Returns

ADMA error status flags mask(_usdhc_adma_error_status_flag).


```
static inline uint32_t USDHC_GetPresentStatusFlags(USDHC_Type *base)
```

Gets a present status.

This function gets the present USDHC's status except for an interrupt status and an error status.

Parameters

- base – USDHC peripheral base address.

Returns

Present USDHC's status flags mask(`_usdhc_present_status_flag`).

```
void USDHC_GetCapability(USDHC_Type *base, usdhc_capability_t *capability)
```

Gets the capability information.

Parameters

- base – USDHC peripheral base address.
- capability – Structure to save capability information.

```
static inline void USDHC_ForceClockOn(USDHC_Type *base, bool enable)
```

Forces the card clock on.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag

```
uint32_t USDHC_SetSdClock(USDHC_Type *base, uint32_t srcClock_Hz, uint32_t busClock_Hz)
```

Sets the SD bus clock frequency.

Parameters

- base – USDHC peripheral base address.
- srcClock_Hz – USDHC source clock frequency united in Hz.
- busClock_Hz – SD bus clock frequency united in Hz.

Returns

The nearest frequency of busClock_Hz configured for SD bus.

```
bool USDHC_SetCardActive(USDHC_Type *base, uint32_t timeout)
```

Sends 80 clocks to the card to set it to the active state.

This function must be called each time the card is inserted to ensure that the card can receive the command correctly.

Parameters

- base – USDHC peripheral base address.
- timeout – Timeout to initialize card.

Return values

- true – Set card active successfully.
- false – Set card active failed.

```
static inline void USDHC_AssertHardwareReset(USDHC_Type *base, bool high)
```

Triggers a hardware reset.

Parameters

- base – USDHC peripheral base address.
- high – 1 or 0 level

static inline void USDHC_SetDataBusWidth(USDHC_Type *base, *usdhc_data_bus_width_t* width)
Sets the data transfer width.

Parameters

- base – USDHC peripheral base address.
- width – Data transfer width.

static inline void USDHC_WriteData(USDHC_Type *base, uint32_t data)
Fills the data port.

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

- base – USDHC peripheral base address.
- data – The data about to be sent.

static inline uint32_t USDHC_ReadData(USDHC_Type *base)
Retrieves the data from the data port.

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

- base – USDHC peripheral base address.

Returns

The data has been read.

void USDHC_SendCommand(USDHC_Type *base, *usdhc_command_t* *command)
Sends command function.

Parameters

- base – USDHC peripheral base address.
- command – configuration

static inline void USDHC_EnableWakeupEvent(USDHC_Type *base, uint32_t mask, bool enable)
Enables or disables a wakeup event in low-power mode.

Parameters

- base – USDHC peripheral base address.
- mask – Wakeup events mask(*_usdhc_wakeup_event*).
- enable – True to enable, false to disable.

static inline void USDHC_CardDetectByData3(USDHC_Type *base, bool enable)
Detects card insert status.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag

static inline bool USDHC_DetectCardInsert(USDHC_Type *base)
Detects card insert status.

Parameters

- base – USDHC peripheral base address.

static inline void USDHC_EnableSdioControl(USDHC_Type *base, uint32_t mask, bool enable)
Enables or disables the SDIO card control.

Parameters

- base – USDHC peripheral base address.
- mask – SDIO card control flags mask(_usdhc_sdio_control_flag).
- enable – True to enable, false to disable.

static inline void USDHC_SetContinueRequest(USDHC_Type *base)
Restarts a transaction which has stopped at the block GAP for the SDIO card.

Parameters

- base – USDHC peripheral base address.

static inline void USDHC_RequestStopAtBlockGap(USDHC_Type *base, bool enable)
Request stop at block gap function.

Parameters

- base – USDHC peripheral base address.
- enable – True to stop at block gap, false to normal transfer.

void USDHC_SetMmcBootConfig(USDHC_Type *base, const *usdhc_boot_config_t* *config)
Configures the MMC boot feature.

Example:

```
usdhc_boot_config_t config;  
config.ackTimeoutCount = 4;  
config.bootMode = kUSDHC_BootModeNormal;  
config.blockCount = 5;  
config.enableBootAck = true;  
config.enableBoot = true;  
config.enableAutoStopAtBlockGap = true;  
USDHC_SetMmcBootConfig(USDHC, &config);
```

Parameters

- base – USDHC peripheral base address.
- config – The MMC boot configuration information.

static inline void USDHC_EnableMmcBoot(USDHC_Type *base, bool enable)
Enables or disables the mmc boot mode.

Parameters

- base – USDHC peripheral base address.
- enable – True to enable, false to disable.

static inline void USDHC_SetForceEvent(USDHC_Type *base, uint32_t mask)
Forces generating events according to the given mask.

Parameters

- base – USDHC peripheral base address.
- mask – The force events bit position (_usdhc_force_event).

static inline bool USDHC_RequestTuningForSDR50(USDHC_Type *base)
Checks the SDR50 mode request tuning bit. When this bit set, application shall perform tuning for SDR50 mode.

Parameters

- base – USDHC peripheral base address.

static inline bool USDHC_RequestReTuning(USDHC_Type *base)

Checks the request re-tuning bit. When this bit is set, user should do manual tuning or standard tuning function.

Parameters

- base – USDHC peripheral base address.

static inline void USDHC_EnableAutoTuning(USDHC_Type *base, bool enable)

The SDR104 mode auto tuning enable and disable. This function should be called after tuning function execute pass, auto tuning will handle by hardware.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag

void USDHC_EnableAutoTuningForCmdAndData(USDHC_Type *base)

The auto tuning enable for CMD/DATA line.

Parameters

- base – USDHC peripheral base address.

void USDHC_EnableManualTuning(USDHC_Type *base, bool enable)

Manual tuning trigger or abort. User should handle the tuning cmd and find the boundary of the delay then calculate a average value which will be configured to the **CLK_TUNE_CTRL_STATUS**. This function should be called before function **USDHC_AdjustDelayForManualTuning**.

Parameters

- base – USDHC peripheral base address.
- enable – tuning enable flag

static inline uint32_t USDHC_GetTuningDelayStatus(USDHC_Type *base)

Get the tuning delay cell setting.

Parameters

- base – USDHC peripheral base address.

Return values

CLK – Tuning Control and Status register value.

status_t USDHC_SetTuningDelay(USDHC_Type *base, uint32_t preDelay, uint32_t outDelay, uint32_t postDelay)

The tuning delay cell setting.

Parameters

- base – USDHC peripheral base address.
- preDelay – Set the number of delay cells on the feedback clock between the feedback clock and CLK_PRE.
- outDelay – Set the number of delay cells on the feedback clock between CLK_PRE and CLK_OUT.
- postDelay – Set the number of delay cells on the feedback clock between CLK_OUT and CLK_POST.

Return values

- kStatus_Fail – config the delay setting fail
- kStatus_Success – config the delay setting success

`status_t USDHC_AdjustDelayForManualTuning(USDHC_Type *base, uint32_t delay)`
Adjusts delay for mannual tuning.

Deprecated:

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

Parameters

- base – USDHC peripheral base address.
- delay – setting configuration

Return values

- kStatus_Fail – config the delay setting fail
- kStatus_Success – config the delay setting success

`static inline void USDHC_SetStandardTuningCounter(USDHC_Type *base, uint8_t counter)`
set tuning counter tuning.

Parameters

- base – USDHC peripheral base address.
- counter – tuning counter

Return values

- kStatus_Fail – config the delay setting fail
- kStatus_Success – config the delay setting success

`void USDHC_EnableStandardTuning(USDHC_Type *base, uint32_t tuningStartTap, uint32_t step, bool enable)`

The enable standard tuning function. The standard tuning window and tuning counter using the default config tuning cmd is sent by the software, user need to check whether the tuning result can be used for SDR50, SDR104, and HS200 mode tuning.

Parameters

- base – USDHC peripheral base address.
- tuningStartTap – start tap
- step – tuning step
- enable – enable/disable flag

`static inline uint32_t USDHC_GetExecuteStdTuningStatus(USDHC_Type *base)`
Gets execute STD tuning status.

Parameters

- base – USDHC peripheral base address.

`static inline uint32_t USDHC_CheckStdTuningResult(USDHC_Type *base)`
Checks STD tuning result.

Parameters

- base – USDHC peripheral base address.

static inline uint32_t USDHC_CheckTuningError(USDHC_Type *base)

Checks tuning error.

Parameters

- base – USDHC peripheral base address.

void USDHC_EnableDDRMMode(USDHC_Type *base, bool enable, uint32_t nibblePos)

The enable/disable DDR mode.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag
- nibblePos – nibble position

static inline void USDHC_EnableHS400Mode(USDHC_Type *base, bool enable)

The enable/disable HS400 mode.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag

static inline void USDHC_ResetStrobeDLL(USDHC_Type *base)

Resets the strobe DLL.

Parameters

- base – USDHC peripheral base address.

static inline void USDHC_EnableStrobeDLL(USDHC_Type *base, bool enable)

Enables/disables the strobe DLL.

Parameters

- base – USDHC peripheral base address.
- enable – enable/disable flag

void USDHC_ConfigStrobeDLL(USDHC_Type *base, uint32_t delayTarget, uint32_t updateInterval)

Configures the strobe DLL delay target and update interval.

Parameters

- base – USDHC peripheral base address.
- delayTarget – delay target
- updateInterval – update interval

static inline void USDHC_SetStrobeDllOverride(USDHC_Type *base, uint32_t delayTaps)

Enables manual override for slave delay chain using **STROBE_SLV_OVERRIDE_VAL**.

Parameters

- base – USDHC peripheral base address.
- delayTaps – Valid delay taps range from 1 - 128 taps. A value of 0 selects tap 1, and a value of 0x7F selects tap 128.

static inline uint32_t USDHC_GetStrobeDLLStatus(USDHC_Type *base)

Gets the strobe DLL status.

Parameters

- base – USDHC peripheral base address.

```
void USDHC_SetDataConfig(USDHC_Type *base, usdhc_transfer_direction_t dataDirection,  
                        uint32_t blockCount, uint32_t blockSize)
```

USDHC data configuration.

Parameters

- base – USDHC peripheral base address.
- dataDirection – Data direction, tx or rx.
- blockCount – Data block count.
- blockSize – Data block size.

```
void USDHC_TransferCreateHandle(USDHC_Type *base, usdhc_handle_t *handle, const  
                               usdhc_transfer_callback_t *callback, void *userData)
```

Creates the USDHC handle.

Parameters

- base – USDHC peripheral base address.
- handle – USDHC handle pointer.
- callback – Structure pointer to contain all callback functions.
- userData – Callback function parameter.

```
status_t USDHC_TransferNonBlocking(USDHC_Type *base, usdhc_handle_t *handle,  
                                   usdhc_adma_config_t *dmaConfig, usdhc_transfer_t  
                                   *transfer)
```

Transfers the command/data using an interrupt and an asynchronous method.

This function sends a command and data and returns immediately. It doesn't wait for the transfer to complete or to encounter an error. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

Note: Call API USDHC_TransferCreateHandle when calling this API.

Parameters

- base – USDHC peripheral base address.
- handle – USDHC handle.
- dmaConfig – ADMA configuration.
- transfer – Transfer content.

Return values

- kStatus_InvalidArgument – Argument is invalid.
- kStatus_USDHC_BusyTransferring – Busy transferring.
- kStatus_USDHC_PrepareAdmaDescriptorFailed – Prepare ADMA descriptor failed.
- kStatus_Success – Operate successfully.

```
status_t USDHC_TransferBlocking(USDHC_Type *base, usdhc_adma_config_t *dmaConfig,  
                               usdhc_transfer_t *transfer)
```

Transfers the command/data using a blocking method.

This function waits until the command response/data is received or the USDHC encounters an error by polling the status flag.

The application must not call this API in multiple threads at the same time. Because this API doesn't support the re-entry mechanism.

Note: There is no need to call API USDHC_TransferCreateHandle when calling this API.

Parameters

- base – USDHC peripheral base address.
- dmaConfig – adma configuration
- transfer – Transfer content.

Return values

- kStatus_InvalidArgument – Argument is invalid.
- kStatus_USDHC_PrepareAdmaDescriptorFailed – Prepare ADMA descriptor failed.
- kStatus_USDHC_SendCommandFailed – Send command failed.
- kStatus_USDHC_TransferDataFailed – Transfer data failed.
- kStatus_Success – Operate successfully.

void USDHC_TransferHandleIRQ(USDHC_Type *base, *usdhc_handle_t* *handle)
IRQ handler for the USDHC.

This function deals with the IRQs on the given host controller.

Parameters

- base – USDHC peripheral base address.
- handle – USDHC handle.

FSL_USDHC_DRIVER_VERSION
Driver version 2.8.5.

Enum_usdhc_status. USDHC status.

Values:

enumerator kStatus_USDHC_BusyTransferring
Transfer is on-going.

enumerator kStatus_USDHC_PrepareAdmaDescriptorFailed
Set DMA descriptor failed.

enumerator kStatus_USDHC_SendCommandFailed
Send command failed.

enumerator kStatus_USDHC_TransferDataFailed
Transfer data failed.

enumerator kStatus_USDHC_DMADDataAddrNotAlign
Data address not aligned.

enumerator kStatus_USDHC_ReTuningRequest
Re-tuning request.

enumerator kStatus_USDHC_TuningError

Tuning error.

enumerator kStatus_USDHC_NotSupport

Not support.

enumerator kStatus_USDHC_TransferDataComplete

Transfer data complete.

enumerator kStatus_USDHC_SendCommandSuccess

Transfer command complete.

enumerator kStatus_USDHC_TransferDMAComplete

Transfer DMA complete.

Enum _usdhc_capability_flag. Host controller capabilities flag mask. .

Values:

enumerator kUSDHC_SupportAdmaFlag

Support ADMA.

enumerator kUSDHC_SupportHighSpeedFlag

Support high-speed.

enumerator kUSDHC_SupportDmaFlag

Support DMA.

enumerator kUSDHC_SupportSuspendResumeFlag

Support suspend/resume.

enumerator kUSDHC_SupportV330Flag

Support voltage 3.3V.

enumerator kUSDHC_SupportV300Flag

Support voltage 3.0V.

enumerator kUSDHC_Support4BitFlag

Flag in HTCAPBLT_MBL's position, supporting 4-bit mode.

enumerator kUSDHC_Support8BitFlag

Flag in HTCAPBLT_MBL's position, supporting 8-bit mode.

enumerator kUSDHC_SupportDDR50Flag

SD version 3.0 new feature, supporting DDR50 mode.

enumerator kUSDHC_SupportSDR104Flag

Support SDR104 mode.

enumerator kUSDHC_SupportSDR50Flag

Support SDR50 mode.

Enum _usdhc_wakeup_event. Wakeup event mask. .

Values:

enumerator kUSDHC_WakeupEventOnCardInt

Wakeup on card interrupt.

enumerator kUSDHC_WakeupEventOnCardInsert

Wakeup on card insertion.

enumerator kUSDHC_WakeupEventOnCardRemove
Wakeup on card removal.

enumerator kUSDHC_WakeupEventsAll
All wakeup events

Enum _usdhc_reset. Reset type mask. .

Values:

enumerator kUSDHC_ResetAll
Reset all except card detection.

enumerator kUSDHC_ResetCommand
Reset command line.

enumerator kUSDHC_ResetData
Reset data line.

enumerator kUSDHC_ResetTuning
Reset tuning circuit.

enumerator kUSDHC_ResetsAll
All reset types

Enum _usdhc_transfer_flag. Transfer flag mask.

Values:

enumerator kUSDHC_EnableDmaFlag
Enable DMA.

enumerator kUSDHC_CommandTypeSuspendFlag
Suspend command.

enumerator kUSDHC_CommandTypeResumeFlag
Resume command.

enumerator kUSDHC_CommandTypeAbortFlag
Abort command.

enumerator kUSDHC_EnableBlockCountFlag
Enable block count.

enumerator kUSDHC_EnableAutoCommand12Flag
Enable auto CMD12.

enumerator kUSDHC_DataReadFlag
Enable data read.

enumerator kUSDHC_MultipleBlockFlag
Multiple block data read/write.

enumerator kUSDHC_EnableAutoCommand23Flag
Enable auto CMD23.

enumerator kUSDHC_ResponseLength136Flag
136-bit response length.

enumerator kUSDHC_ResponseLength48Flag
48-bit response length.

enumerator kUSDHC_ResponseLength48BusyFlag
48-bit response length with busy status.

enumerator kUSDHC_EnableCrcCheckFlag
Enable CRC check.

enumerator kUSDHC_EnableIndexCheckFlag
Enable index check.

enumerator kUSDHC_DataPresentFlag
Data present flag.

Enum _usdhc_present_status_flag. Present status flag mask. .

Values:

enumerator kUSDHC_CommandInhibitFlag
Command inhibit.

enumerator kUSDHC_DataInhibitFlag
Data inhibit.

enumerator kUSDHC_DataLineActiveFlag
Data line active.

enumerator kUSDHC_SdClockStableFlag
SD bus clock stable.

enumerator kUSDHC_WriteTransferActiveFlag
Write transfer active.

enumerator kUSDHC_ReadTransferActiveFlag
Read transfer active.

enumerator kUSDHC_BufferWriteEnableFlag
Buffer write enable.

enumerator kUSDHC_BufferReadEnableFlag
Buffer read enable.

enumerator kUSDHC_ReTuningRequestFlag
Re-tuning request flag, only used for SDR104 mode.

enumerator kUSDHC_DelaySettingFinishedFlag
Delay setting finished flag.

enumerator kUSDHC_CardInsertedFlag
Card inserted.

enumerator kUSDHC_CommandLineLevelFlag
Command line signal level.

enumerator kUSDHC_Data0LineLevelFlag
Data0 line signal level.

enumerator kUSDHC_Data1LineLevelFlag
Data1 line signal level.

enumerator kUSDHC_Data2LineLevelFlag
Data2 line signal level.

enumerator kUSDHC_Data3LineLevelFlag
Data3 line signal level.

enumerator kUSDHC_Data4LineLevelFlag
Data4 line signal level.

enumerator kUSDHC_Data5LineLevelFlag
Data5 line signal level.

enumerator kUSDHC_Data6LineLevelFlag
Data6 line signal level.

enumerator kUSDHC_Data7LineLevelFlag
Data7 line signal level.

Enum _usdhc_interrupt_status_flag. Interrupt status flag mask. .

Values:

enumerator kUSDHC_CommandCompleteFlag
Command complete.

enumerator kUSDHC_DataCompleteFlag
Data complete.

enumerator kUSDHC_BlockGapEventFlag
Block gap event.

enumerator kUSDHC_DmaCompleteFlag
DMA interrupt.

enumerator kUSDHC_BufferWriteReadyFlag
Buffer write ready.

enumerator kUSDHC_BufferReadReadyFlag
Buffer read ready.

enumerator kUSDHC_CardInsertionFlag
Card inserted.

enumerator kUSDHC_CardRemovalFlag
Card removed.

enumerator kUSDHC_CardInterruptFlag
Card interrupt.

enumerator kUSDHC_ReTuningEventFlag
Re-Tuning event, only for SD3.0 SDR104 mode.

enumerator kUSDHC_TuningPassFlag
SDR104 mode tuning pass flag.

enumerator kUSDHC_TuningErrorFlag
SDR104 tuning error flag.

enumerator kUSDHC_CommandTimeoutFlag
Command timeout error.

enumerator kUSDHC_CommandCrcErrorFlag
Command CRC error.

enumerator kUSDHC_CommandEndBitErrorFlag

Command end bit error.

enumerator kUSDHC_CommandIndexErrorFlag

Command index error.

enumerator kUSDHC_DataTimeoutFlag

Data timeout error.

enumerator kUSDHC_DataCrcErrorFlag

Data CRC error.

enumerator kUSDHC_DataEndBitErrorFlag

Data end bit error.

enumerator kUSDHC_AutoCommand12ErrorFlag

Auto CMD12 error.

enumerator kUSDHC_DmaErrorFlag

DMA error.

enumerator kUSDHC_CommandErrorFlag

Command error

enumerator kUSDHC_DataErrorFlag

Data error

enumerator kUSDHC_ErrorFlag

All error

enumerator kUSDHC_DataFlag

Data interrupts

enumerator kUSDHC_DataDMAFlag

Data interrupts

enumerator kUSDHC_CommandFlag

Command interrupts

enumerator kUSDHC_CardDetectFlag

Card detection interrupts

enumerator kUSDHC_SDR104TuningFlag

SDR104 tuning flag.

enumerator kUSDHC_AllInterruptFlags

All flags mask

Enum _usdhc_auto_command12_error_status_flag. Auto CMD12 error status flag mask. .

Values:

enumerator kUSDHC_AutoCommand12NotExecutedFlag

Not executed error.

enumerator kUSDHC_AutoCommand12TimeoutFlag

Timeout error.

enumerator kUSDHC_AutoCommand12EndBitErrorFlag

End bit error.

enumerator kUSDHC__AutoCommand12CrcErrorFlag
CRC error.

enumerator kUSDHC__AutoCommand12IndexErrorFlag
Index error.

enumerator kUSDHC__AutoCommand12NotIssuedFlag
Not issued error.

Enum _usdhc_standard_tuning. Standard tuning flag.

Values:

enumerator kUSDHC__ExecuteTuning
Used to start tuning procedure.

enumerator kUSDHC__TuningSampleClockSel
When **std_tuning_en** bit is set, this bit is used to select sampling clock.

Enum _usdhc_adma_error_status_flag. ADMA error status flag mask. .

Values:

enumerator kUSDHC__AdmaLenghMismatchFlag
Length mismatch error.

enumerator kUSDHC__AdmaDescriptorErrorFlag
Descriptor error.

Enum _usdhc_adma_error_state. ADMA error state.

This state is the detail state when ADMA error has occurred.

Values:

enumerator kUSDHC__AdmaErrorStateStopDma
Stop DMA, previous location set in the ADMA system address is errored address.

enumerator kUSDHC__AdmaErrorStateFetchDescriptor
Fetch descriptor, current location set in the ADMA system address is errored address.

enumerator kUSDHC__AdmaErrorStateChangeAddress
Change address, no DMA error has occurred.

enumerator kUSDHC__AdmaErrorStateTransferData
Transfer data, previous location set in the ADMA system address is errored address.

enumerator kUSDHC__AdmaErrorStateInvalidLength
Invalid length in ADMA descriptor.

enumerator kUSDHC__AdmaErrorStateInvalidDescriptor
Invalid descriptor fetched by ADMA.

enumerator kUSDHC__AdmaErrorState
ADMA error state

Enum _usdhc_force_event. Force event bit position. .

Values:

enumerator kUSDHC_ForceEventAutoCommand12NotExecuted
Auto CMD12 not executed error.

enumerator kUSDHC_ForceEventAutoCommand12Timeout
Auto CMD12 timeout error.

enumerator kUSDHC_ForceEventAutoCommand12CrcError
Auto CMD12 CRC error.

enumerator kUSDHC_ForceEventEndBitError
Auto CMD12 end bit error.

enumerator kUSDHC_ForceEventAutoCommand12IndexError
Auto CMD12 index error.

enumerator kUSDHC_ForceEventAutoCommand12NotIssued
Auto CMD12 not issued error.

enumerator kUSDHC_ForceEventCommandTimeout
Command timeout error.

enumerator kUSDHC_ForceEventCommandCrcError
Command CRC error.

enumerator kUSDHC_ForceEventCommandEndBitError
Command end bit error.

enumerator kUSDHC_ForceEventCommandIndexError
Command index error.

enumerator kUSDHC_ForceEventDataTimeout
Data timeout error.

enumerator kUSDHC_ForceEventDataCrcError
Data CRC error.

enumerator kUSDHC_ForceEventDataEndBitError
Data end bit error.

enumerator kUSDHC_ForceEventAutoCommand12Error
Auto CMD12 error.

enumerator kUSDHC_ForceEventCardInt
Card interrupt.

enumerator kUSDHC_ForceEventDmaError
Dma error.

enumerator kUSDHC_ForceEventTuningError
Tuning error.

enumerator kUSDHC_ForceEventsAll
All force event flags mask.

enum __usdhc_transfer_direction
Data transfer direction.

Values:

enumerator kUSDHC_TransferDirectionReceive
USDHC transfer direction receive.

enumerator kUSDHC_TransferDirectionSend
USDHC transfer direction send.

enum __usdhc_data_bus_width
Data transfer width.

Values:

enumerator kUSDHC_DataBusWidth1Bit
1-bit mode

enumerator kUSDHC_DataBusWidth4Bit
4-bit mode

enumerator kUSDHC_DataBusWidth8Bit
8-bit mode

enum __usdhc_endian_mode
Endian mode.

Values:

enumerator kUSDHC_EndianModeBig
Big endian mode.

enumerator kUSDHC_EndianModeHalfWordBig
Half word big endian mode.

enumerator kUSDHC_EndianModeLittle
Little endian mode.

enum __usdhc_dma_mode
DMA mode.

Values:

enumerator kUSDHC_DmaModeSimple
External DMA.

enumerator kUSDHC_DmaModeAdma1
ADMA1 is selected.

enumerator kUSDHC_DmaModeAdma2
ADMA2 is selected.

enumerator kUSDHC_ExternalDMA
External DMA mode selected.

Enum __usdhc_sdio_control_flag. SDIO control flag mask. .

Values:

enumerator kUSDHC_StopAtBlockGapFlag
Stop at block gap.

enumerator kUSDHC_ReadWaitControlFlag
Read wait control.

enumerator kUSDHC_InterruptAtBlockGapFlag
Interrupt at block gap.

enumerator kUSDHC_ReadDoneNo8CLK
Read done without 8 clk for block gap.

enumerator kUSDHC_ExactBlockNumberReadFlag
Exact block number read.

enum __usdhc_boot_mode
MMC card boot mode.

Values:

enumerator kUSDHC_BootModeNormal
Normal boot

enumerator kUSDHC_BootModeAlternative
Alternative boot

enum __usdhc_card_command_type
The command type.

Values:

enumerator kCARD_CommandTypeNormal
Normal command

enumerator kCARD_CommandTypeSuspend
Suspend command

enumerator kCARD_CommandTypeResume
Resume command

enumerator kCARD_CommandTypeAbort
Abort command

enumerator kCARD_CommandTypeEmpty
Empty command

enum __usdhc_card_response_type
The command response type.

Defines the command response type from card to host controller.

Values:

enumerator kCARD_ResponseTypeNone
Response type: none

enumerator kCARD_ResponseTypeR1
Response type: R1

enumerator kCARD_ResponseTypeR1b
Response type: R1b

enumerator kCARD_ResponseTypeR2
Response type: R2

enumerator kCARD_ResponseTypeR3
Response type: R3

enumerator kCARD_ResponseTypeR4
Response type: R4

enumerator kCARD_ResponseTypeR5
Response type: R5

enumerator kCARD_ResponseTypeR5b
Response type: R5b

enumerator kCARD_ResponseTypeR6

Response type: R6

enumerator kCARD_ResponseTypeR7

Response type: R7

Enum `_usdhc_adma1_descriptor_flag`. The mask for the control/status field in ADMA1 descriptor.

Values:

enumerator kUSDHC_Adma1DescriptorValidFlag

Valid flag.

enumerator kUSDHC_Adma1DescriptorEndFlag

End flag.

enumerator kUSDHC_Adma1DescriptorInterruptFlag

Interrupt flag.

enumerator kUSDHC_Adma1DescriptorActivity1Flag

Activity 1 flag.

enumerator kUSDHC_Adma1DescriptorActivity2Flag

Activity 2 flag.

enumerator kUSDHC_Adma1DescriptorTypeNop

No operation.

enumerator kUSDHC_Adma1DescriptorTypeTransfer

Transfer data.

enumerator kUSDHC_Adma1DescriptorTypeLink

Link descriptor.

enumerator kUSDHC_Adma1DescriptorTypeSetLength

Set data length.

Enum `_usdhc_adma2_descriptor_flag`. ADMA1 descriptor control and status mask.

Values:

enumerator kUSDHC_Adma2DescriptorValidFlag

Valid flag.

enumerator kUSDHC_Adma2DescriptorEndFlag

End flag.

enumerator kUSDHC_Adma2DescriptorInterruptFlag

Interrupt flag.

enumerator kUSDHC_Adma2DescriptorActivity1Flag

Activity 1 mask.

enumerator kUSDHC_Adma2DescriptorActivity2Flag

Activity 2 mask.

enumerator kUSDHC_Adma2DescriptorTypeNop

No operation.

enumerator kUSDHC_Adma2DescriptorTypeReserved

Reserved.

enumerator kUSDHC_Adma2DescriptorTypeTransfer

Transfer type.

enumerator kUSDHC_Adma2DescriptorTypeLink

Link type.

Enum _usdhc_adma_flag. ADMA descriptor configuration flag. .

Values:

enumerator kUSDHC_AdmaDescriptorSingleFlag

Try to finish the transfer in a single ADMA descriptor. If transfer size is bigger than one ADMA descriptor's ability, new another descriptor for data transfer.

enumerator kUSDHC_AdmaDescriptorMultipleFlag

Create multiple ADMA descriptors within the ADMA table, this is used for mmc boot mode specifically, which need to modify the ADMA descriptor on the fly, so the flag should be used combining with stop at block gap feature.

enum _usdhc_burst_len

DMA transfer burst len config.

Values:

enumerator kUSDHC_EnBurstLenForINCR

Enable burst len for INCR.

enumerator kUSDHC_EnBurstLenForINCR4816

Enable burst len for INCR4/INCR8/INCR16.

enumerator kUSDHC_EnBurstLenForINCR4816WRAP

Enable burst len for INCR4/8/16 WRAP.

Enum _usdhc_transfer_data_type. Transfer data type definition.

Values:

enumerator kUSDHC_TransferDataNormal

Transfer normal read/write data.

enumerator kUSDHC_TransferDataTuning

Transfer tuning data.

enumerator kUSDHC_TransferDataBoot

Transfer boot data.

enumerator kUSDHC_TransferDataBootcontinuous

Transfer boot data continuously.

typedef enum _usdhc_transfer_direction usdhc_transfer_direction_t

Data transfer direction.

typedef enum _usdhc_data_bus_width usdhc_data_bus_width_t

Data transfer width.

typedef enum _usdhc_endian_mode usdhc_endian_mode_t

Endian mode.

typedef enum _usdhc_dma_mode usdhc_dma_mode_t

DMA mode.

```
typedef enum _usdhc_boot_mode usdhc_boot_mode_t
```

MMC card boot mode.

```
typedef enum _usdhc_card_command_type usdhc_card_command_type_t
```

The command type.

```
typedef enum _usdhc_card_response_type usdhc_card_response_type_t
```

The command response type.

Defines the command response type from card to host controller.

```
typedef enum _usdhc_burst_len usdhc_burst_len_t
```

DMA transfer burst len config.

```
typedef uint32_t usdhc_adma1_descriptor_t
```

Defines the ADMA1 descriptor structure.

```
typedef struct _usdhc_adma2_descriptor usdhc_adma2_descriptor_t
```

Defines the ADMA2 descriptor structure.

```
typedef struct _usdhc_capability usdhc_capability_t
```

USDHC capability information.

Defines a structure to save the capability information of USDHC.

```
typedef struct _usdhc_boot_config usdhc_boot_config_t
```

Data structure to configure the MMC boot feature.

```
typedef struct _usdhc_config usdhc_config_t
```

Data structure to initialize the USDHC.

```
typedef struct _usdhc_command usdhc_command_t
```

Card command descriptor.

Defines card command-related attribute.

```
typedef struct _usdhc_adma_config usdhc_adma_config_t
```

ADMA configuration.

```
typedef struct _usdhc_scatter_gather_data_list usdhc_scatter_gather_data_list_t
```

Card scatter gather data list.

Allow application register uncontinuous data buffer for data transfer.

```
typedef struct _usdhc_scatter_gather_data usdhc_scatter_gather_data_t
```

Card scatter gather data descriptor.

Defines a structure to contain data-related attribute. The 'enableIgnoreError' is used when upper card driver wants to ignore the error event to read/write all the data and not to stop read/write immediately when an error event happens. For example, bus testing procedure for MMC card.

```
typedef struct _usdhc_scatter_gather_transfer usdhc_scatter_gather_transfer_t
```

usdhc scatter gather transfer.

```
typedef struct _usdhc_data usdhc_data_t
```

Card data descriptor.

Defines a structure to contain data-related attribute. The 'enableIgnoreError' is used when upper card driver wants to ignore the error event to read/write all the data and not to stop read/write immediately when an error event happens. For example, bus testing procedure for MMC card.

```
typedef struct _usdhc_transfer usdhc_transfer_t
```

Transfer state.

```
typedef struct _usdhc_handle usdhc_handle_t
```

USDHC handle typedef.

```
typedef struct _usdhc_transfer_callback usdhc_transfer_callback_t
```

USDHC callback functions.

```
typedef status_t (*usdhc_transfer_function_t)(USDHC_Type *base, usdhc_transfer_t *content)
```

USDHC transfer function.

```
typedef struct _usdhc_host usdhc_host_t
```

USDHC host descriptor.

```
USDHC_MAX_BLOCK_COUNT
```

Maximum block count can be set one time.

```
FSL_USDHC_ENABLE_SCATTER_GATHER_TRANSFER
```

USDHC scatter gather feature control macro.

```
USDHC_ADMA1_ADDRESS_ALIGN
```

The alignment size for ADDRESS filed in ADMA1's descriptor.

```
USDHC_ADMA1_LENGTH_ALIGN
```

The alignment size for LENGTH field in ADMA1's descriptor.

```
USDHC_ADMA2_ADDRESS_ALIGN
```

The alignment size for ADDRESS field in ADMA2's descriptor.

```
USDHC_ADMA2_LENGTH_ALIGN
```

The alignment size for LENGTH filed in ADMA2's descriptor.

```
USDHC_ADMA1_DESCRIPTOR_ADDRESS_SHIFT
```

The bit shift for ADDRESS filed in ADMA1's descriptor.

Address/page field	Reserved	Attribute					
31 12	11 6	05	04	03	02	01	00
address or data length	000000	Act2	Act1	0	Int	End	Valid

Act2	Act1	Comment	31-28	27-12
0	0	No op	Don't care	
0	1	Set data length	0000	Data Length
1	0	Transfer data	Data address	
1	1	Link descriptor	Descriptor address	

```
USDHC_ADMA1_DESCRIPTOR_ADDRESS_MASK
```

The bit mask for ADDRESS field in ADMA1's descriptor.

```
USDHC_ADMA1_DESCRIPTOR_LENGTH_SHIFT
```

The bit shift for LENGTH filed in ADMA1's descriptor.

```
USDHC_ADMA1_DESCRIPTOR_LENGTH_MASK
```

The mask for LENGTH field in ADMA1's descriptor.

USDHC_ADMA1_DESCRIPTOR_MAX_LENGTH_PER_ENTRY

The maximum value of LENGTH filed in ADMA1's descriptor. Since the max transfer size ADMA1 support is 65535 which is indivisible by 4096, so to make sure a large data load transfer (>64KB) continuously (require the data address be always align with 4096), software will set the maximum data length for ADMA1 to (64 - 4)KB.

USDHC_ADMA2_DESCRIPTOR_LENGTH_SHIFT

The bit shift for LENGTH field in ADMA2's descriptor.

Address field	Length	Reserved	Attribute						
63 32	31 16	15 06	05	04	03	02	01	00	
32-bit address	16-bit length	0000000000	Act2	Act1	0	Int	End	Valid	

Act2	Act1	Comment	Operation
0	0	No op	Don't care
0	1	Reserved	Read this line and go to next one
1	0	Transfer data	Transfer data with address and length set in this descriptor line
1	1	Link descriptor	Link to another descriptor

USDHC_ADMA2_DESCRIPTOR_LENGTH_MASK

The bit mask for LENGTH field in ADMA2's descriptor.

USDHC_ADMA2_DESCRIPTOR_MAX_LENGTH_PER_ENTRY

The maximum value of LENGTH field in ADMA2's descriptor.

struct __usdhc_adma2_descriptor

#include <fsl_usdhc.h> Defines the ADMA2 descriptor structure.

Public Members

uint32_t attribute

The control and status field.

uint32_t address

The address field.

struct __usdhc_capability

#include <fsl_usdhc.h> USDHC capability information.

Defines a structure to save the capability information of USDHC.

Public Members

uint32_t sdVersion

Support SD card/sdio version.

uint32_t mmcVersion

Support EMMC card version.

uint32_t maxBlockLength

Maximum block length united as byte.

uint32_t maxBlockCount

Maximum block count can be set one time.

uint32_t flags

Capability flags to indicate the support information(_usdhc_capability_flag).

struct __usdhc_boot_config

#include <fsl_usdhc.h> Data structure to configure the MMC boot feature.

Public Members

uint32_t ackTimeoutCount

Timeout value for the boot ACK. The available range is 0 ~ 15.

usdhc_boot_mode_t bootMode

Boot mode selection.

uint32_t blockCount

Stop at block gap value of automatic mode. Available range is 0 ~ 65535.

size_t blockSize

Block size.

bool enableBootAck

Enable or disable boot ACK.

bool enableAutoStopAtBlockGap

Enable or disable auto stop at block gap function in boot period.

struct __usdhc_config

#include <fsl_usdhc.h> Data structure to initialize the USDHC.

Public Members

uint32_t dataTimeout

Data timeout value.

usdhc_endian_mode_t endianMode

Endian mode.

uint8_t readWatermarkLevel

Watermark level for DMA read operation. Available range is 1 ~ 128.

uint8_t writeWatermarkLevel

Watermark level for DMA write operation. Available range is 1 ~ 128.

struct __usdhc_command

#include <fsl_usdhc.h> Card command descriptor.

Defines card command-related attribute.

Public Members

uint32_t index

Command index.

uint32_t argument

Command argument.

usdhc_card_command_type_t type

Command type.

usdhc_card_response_type_t responseType

Command response type.

uint32_t response[4U]

Response for this command.

uint32_t responseErrorFlags

Response error flag, which need to check the command reponse.

uint32_t flags

Cmd flags.

struct __usdhc_adma_config

#include <fsl_usdhc.h> ADMA configuration.

Public Members

usdhc_dma_mode_t dmaMode

DMA mode.

uint32_t *admaTable

ADMA table address, can't be null if transfer way is ADMA1/ADMA2.

uint32_t admaTableWords

ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.

struct __usdhc_scatter_gather_data_list

#include <fsl_usdhc.h> Card scatter gather data list.

Allow application register uncontinuous data buffer for data transfer.

struct __usdhc_scatter_gather_data

#include <fsl_usdhc.h> Card scatter gather data descriptor.

Defines a structure to contain data-related attribute. The 'enableIgnoreError' is used when upper card driver wants to ignore the error event to read/write all the data and not to stop read/write immediately when an error event happens. For example, bus testing procedure for MMC card.

Public Members

bool enableAutoCommand12

Enable auto CMD12.

bool enableAutoCommand23

Enable auto CMD23.

bool enableIgnoreError

Enable to ignore error event to read/write all the data.

usdhc_transfer_direction_t dataDirection

data direction

uint8_t dataType

this is used to distinguish the normal/tuning/boot data.

size_t blockSize

Block size.

usdhc_scatter_gather_data_list_t sgData

scatter gather data

struct __usdhc_scatter_gather_transfer

#include <fsl_usdhc.h> usdhc scatter gather transfer.

Public Members

usdhc_scatter_gather_data_t *data

Data to transfer.

usdhc_command_t *command

Command to send.

struct __usdhc_data

#include <fsl_usdhc.h> Card data descriptor.

Defines a structure to contain data-related attribute. The 'enableIgnoreError' is used when upper card driver wants to ignore the error event to read/write all the data and not to stop read/write immediately when an error event happens. For example, bus testing procedure for MMC card.

Public Members

bool enableAutoCommand12

Enable auto CMD12.

bool enableAutoCommand23

Enable auto CMD23.

bool enableIgnoreError

Enable to ignore error event to read/write all the data.

uint8_t dataType

this is used to distinguish the normal/tuning/boot data.

size_t blockSize

Block size.

uint32_t blockCount

Block count.

uint32_t *rxData

Buffer to save data read.

const uint32_t *txData

Data buffer to write.

struct __usdhc_transfer

#include <fsl_usdhc.h> Transfer state.

Public Members

usdhc_data_t *data

Data to transfer.

usdhc_command_t *command
Command to send.

struct __usdhc_transfer_callback
#include <fsl_usdhc.h> USDHC callback functions.

Public Members

void (*CardInserted)(USDHC_Type *base, void *userData)
Card inserted occurs when DAT3/CD pin is for card detect

void (*CardRemoved)(USDHC_Type *base, void *userData)
Card removed occurs

void (*SdioInterrupt)(USDHC_Type *base, void *userData)
SDIO card interrupt occurs

void (*BlockGap)(USDHC_Type *base, void *userData)
stopped at block gap event

void (*TransferComplete)(USDHC_Type *base, *usdhc_handle_t* *handle, *status_t* status, void *userData)
Transfer complete callback.

void (*ReTuning)(USDHC_Type *base, void *userData)
Handle the re-tuning.

struct __usdhc_handle
#include <fsl_usdhc.h> USDHC handle.
Defines the structure to save the USDHC state information and callback function.

Note: All the fields except interruptFlags and transferredWords must be allocated by the user.

Public Members

usdhc_data_t *volatile data
Transfer parameter. Data to transfer.

usdhc_command_t *volatile command
Transfer parameter. Command to send.

volatile uint32_t transferredWords
Transfer status. Words transferred by DATAPORT way.

usdhc_transfer_callback_t callback
Callback function.

void *userData
Parameter for transfer complete callback.

struct __usdhc_host
#include <fsl_usdhc.h> USDHC host descriptor.

Public Members

USDHC_Type *base
USDHC peripheral base address.

uint32_t sourceClock_Hz
USDHC source clock frequency united in Hz.

usdhc_config_t config
USDHC configuration.

usdhc_capability_t capability
USDHC capability information.

usdhc_transfer_function_t transfer
USDHC transfer function.

2.82 WDOG32: 32-bit Watchdog Timer

void WDOG32_GetDefaultConfig(*wdog32_config_t* *config)

Initializes the WDOG32 configuration structure.

This function initializes the WDOG32 configuration structure to default values. The default values are:

```
wdog32Config->enableWdog32 = true;
wdog32Config->clockSource = kWDOG32_ClockSource1;
wdog32Config->prescaler = kWDOG32_ClockPrescalerDivide1;
wdog32Config->workMode.enableWait = true;
wdog32Config->workMode.enableStop = false;
wdog32Config->workMode.enableDebug = false;
wdog32Config->testMode = kWDOG32_TestModeDisabled;
wdog32Config->enableUpdate = true;
wdog32Config->enableInterrupt = false;
wdog32Config->enableWindowMode = false;
wdog32Config->windowValue = 0U;
wdog32Config->timeoutValue = 0xFFFFU;
```

See also:

wdog32_config_t

Parameters

- config – Pointer to the WDOG32 configuration structure.

AT_QUICKACCESS_SECTION_CODE (status_t WDOG32_Init(WDOG_Type *base, const *wdog32_config_t* *config))

Initializes the WDOG32 module.

This function initializes the WDOG32. To reconfigure the WDOG32 without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
wdog32_config_t config;
WDOG32_GetDefaultConfig(&config);
config.timeoutValue = 0x7fU;
config.enableUpdate = true;
WDOG32_Init(wdog_base,&config);
```

Note: If there is errata ERR010536 (FSL_FEATURE_WDOG_HAS_ERRATA_010536 defined as 1), then after calling this function, user need delay at least 4 LPO clock cycles before accessing other WDOG32 registers.

Parameters

- base – WDOG32 peripheral base address.
- config – The configuration of the WDOG32.

Return values

- kStatus_Success – The initialization was successful
- kStatus_Timeout – The initialization timed out

`status_t WDOG32_Deinit(WDOG_Type *base)`

De-initializes the WDOG32 module.

This function shuts down the WDOG32. Ensure that the WDOG_CS.UPDATE is 1, which means that the register update is enabled.

Parameters

- base – WDOG32 peripheral base address.

Return values

- kStatus_Success – The de-initialization was successful
- kStatus_Timeout – The de-initialization timed out

`AT_QUICKACCESS_SECTION_CODE (status_t WDOG32_Unlock(WDOG_Type *base))`

Unlocks the WDOG32 register written.

This function unlocks the WDOG32 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG32 peripheral base address

Return values

- kStatus_Success – The unlock sequence was successful
- kStatus_Timeout – The unlock sequence timed out

`AT_QUICKACCESS_SECTION_CODE (void WDOG32_Enable(WDOG_Type *base))`

Enables the WDOG32 module.

Disables the WDOG32 module.

This function writes a value into the WDOG_CS register to enable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

This function writes a value into the WDOG_CS register to disable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure

that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- base – WDOG32 peripheral base address

AT_QUICKACCESS_SECTION_CODE (void WDOG32_EnableInterrupts(WDOG_Type *base, uint32_t mask))

Enables the WDOG32 interrupt.

Clears the WDOG32 flag.

Disables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to enable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Example to clear an interrupt flag:

```
WDOG32_ClearStatusFlags(wdog_base, kWDOG32_InterruptFlag);
```

Parameters

- base – WDOG32 peripheral base address.
- mask – The interrupts to enable. The parameter can be a combination of the following source if defined:
 - kWDOG32_InterruptEnable

This function writes a value into the WDOG_CS register to disable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

- base – WDOG32 peripheral base address.
- mask – The interrupts to disabled. The parameter can be a combination of the following source if defined:
 - kWDOG32_InterruptEnable

This function clears the WDOG32 status flag.

- base – WDOG32 peripheral base address.
- mask – The status flags to clear. The parameter can be any combination of the following values:
 - kWDOG32_InterruptFlag

static inline uint32_t WDOG32_GetStatusFlags(WDOG_Type *base)

Gets the WDOG32 all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;  
status = WDOG32_GetStatusFlags(wdog_base) & kWDOG32_RunningFlag;
```

See also:

`_wdog32_status_flags_t`

- true: related status flag has been set.
- false: related status flag is not set.

Parameters

- base – WDOG32 peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

AT_QUICKACCESS_SECTION_CODE (void WDOG32_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount))

Sets the WDOG32 timeout value.

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. To ensure the reconfiguration fits the timing of WCT, unlock function will be called inline.

Parameters

- base – WDOG32 peripheral base address
- timeoutCount – WDOG32 timeout value, count of WDOG32 clock ticks.

AT_QUICKACCESS_SECTION_CODE (void WDOG32_SetWindowValue(WDOG_Type *base, uint16_t windowValue))

Sets the WDOG32 window value.

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- windowValue – WDOG32 window value.

static inline void WDOG32_Refresh(WDOG_Type *base)

Refreshes the WDOG32 timer.

This function feeds the WDOG32. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG32 peripheral base address

static inline uint16_t WDOG32_GetCounterValue(WDOG_Type *base)

Gets the WDOG32 counter value.

This function gets the WDOG32 counter value.

Parameters

- base – WDOG32 peripheral base address.

Returns

Current WDOG32 counter value.

WDOG_FIRST_WORD_OF_UNLOCK

First word of unlock sequence

WDOG_SECOND_WORD_OF_UNLOCK

Second word of unlock sequence

WDOG_FIRST_WORD_OF_REFRESH

First word of refresh sequence

WDOG_SECOND_WORD_OF_REFRESH

Second word of refresh sequence

FSL_WDOG32_DRIVER_VERSION

WDOG32 driver version.

enum __wdog32_clock_source

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

Values:

enumerator kWDOG32_ClockSource0

Clock source 0

enumerator kWDOG32_ClockSource1

Clock source 1

enumerator kWDOG32_ClockSource2

Clock source 2

enumerator kWDOG32_ClockSource3

Clock source 3

enum __wdog32_clock_prescaler

Describes the selection of the clock prescaler.

Values:

enumerator kWDOG32_ClockPrescalerDivide1

Divided by 1

enumerator kWDOG32_ClockPrescalerDivide256

Divided by 256

enum __wdog32_test_mode

Describes WDOG32 test mode.

Values:

enumerator kWDOG32_TestModeDisabled

Test Mode disabled

enumerator kWDOG32_UserModeEnabled

User Mode enabled

enumerator kWDOG32_LowByteTest

Test Mode enabled, only low byte is used

enumerator kWDOG32_HighByteTest

Test Mode enabled, only high byte is used

enum __wdog32_interrupt_enable_t

WDOG32 interrupt configuration structure.

This structure contains the settings for all of the WDOG32 interrupt configurations.

Values:

enumerator kWDOG32_InterruptEnable

Interrupt is generated before forcing a reset

enum __wdog32_status_flags_t

WDOG32 status flags.

This structure contains the WDOG32 status flags for use in the WDOG32 functions.

Values:

enumerator kWDOG32_RunningFlag

Running flag, set when WDOG32 is enabled

enumerator kWDOG32_InterruptFlag

Interrupt flag, set when interrupt occurs

typedef enum __wdog32_clock_source wdog32_clock_source_t

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

typedef enum __wdog32_clock_prescaler wdog32_clock_prescaler_t

Describes the selection of the clock prescaler.

typedef struct __wdog32_work_mode wdog32_work_mode_t

Defines WDOG32 work mode.

typedef enum __wdog32_test_mode wdog32_test_mode_t

Describes WDOG32 test mode.

typedef struct __wdog32_config wdog32_config_t

Describes WDOG32 configuration structure.

struct __wdog32_work_mode

#include <fsl_wdog32.h> Defines WDOG32 work mode.

Public Members

bool enableWait

Enables or disables WDOG32 in wait mode

bool enableStop

Enables or disables WDOG32 in stop mode

`bool enableDebug`
Enables or disables WDOG32 in debug mode

`struct __wdog32_config`
#include <fsl_wdog32.h> Describes WDOG32 configuration structure.

Public Members

`bool enableWdog32`
Enables or disables WDOG32

`wdog32_clock_source_t clockSource`
Clock source select

`wdog32_clock_prescaler_t prescaler`
Clock prescaler value

`wdog32_work_mode_t workMode`
Configures WDOG32 work mode in debug stop and wait mode

`wdog32_test_mode_t testMode`
Configures WDOG32 test mode

`bool enableUpdate`
Update write-once register enable

`bool enableInterrupt`
Enables or disables WDOG32 interrupt

`bool enableWindowMode`
Enables or disables WDOG32 window mode

`uint16_t windowValue`
Window value

`uint16_t timeoutValue`
Timeout value

2.83 WUU: Wakeup Unit driver

`void WUU_SetExternalWakeUpPinsConfig(WUU_Type *base, uint8_t pinIndex, const wuu_external_wakeup_pin_config_t *config)`

Enables and Configs External WakeUp Pins.

This function enables/disables the external pin as wakeup input. What's more this function configs pins options, including edge detection wakeup event and operate mode.

Parameters

- `base` – MUU peripheral base address.
- `pinIndex` – The index of the external input pin. See Reference Manual for the details.
- `config` – Pointer to `wuu_external_wakeup_pin_config_t` structure.

`void WUU_ClearExternalWakeupPinsConfig(WUU_Type *base, uint8_t pinIndex)`

Disable and clear external wakeup pin settings.

Parameters

- base – MUU peripheral base address.
- pinIndex – The index of the external input pin.

static inline uint32_t WUU_GetExternalWakeUpPinsFlag(WUU_Type *base)

Gets External Wakeup pin flags.

This function return the external wakeup pin flags.

Parameters

- base – WUU peripheral base address.

Returns

Wakeup flags for all external wakeup pins.

static inline void WUU_ClearExternalWakeUpPinsFlag(WUU_Type *base, uint32_t mask)

Clears External WakeUp Pin flags.

This function clears external wakeup pins flags based on the mask.

Parameters

- base – WUU peripheral base address.
- mask – The mask of Wakeup pin index to be cleared.

void WUU_SetInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex,
 wuw_internal_wakeup_module_event_t event)

Config Internal modules' event as the wake up sources.

This function configs the internal modules event as the wake up sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – Select interrupt or DMA/Trigger of the internal module as the wake up source.

void WUU_ClearInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex,
 wuw_internal_wakeup_module_event_t event)

Disable an on-chip internal modules' event as the wakeup sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – The event(interrupt or DMA/trigger) of the internal module to disable.

static inline uint32_t WUU_GetModuleInterruptFlag(WUU_Type *base)

Get wakeup flags for internal wakeup modules.

Parameters

- base – WUU peripheral base address.

Returns

Wakeup flags for all internal wakeup modules.

```
static inline bool WUU_GetInternalWakeupModuleFlag(WUU_Type *base, uint32_t moduleIndex)
```

Gets the internal module wakeup source flag.

This function checks the flag to detect whether the system is woken up by specific on-chip module interrupt.

Parameters

- base – WWU peripheral base address.
- moduleIndex – A module index, which starts from 0.

Returns

True if the specific pin is a wake up source.

```
void WUU_SetPinFilterConfig(WUU_Type *base, uint8_t filterIndex, const  
                           wuu_pin_filter_config_t *config)
```

Configs and Enables Pin filters.

This function configs Pin filter, including pin select, filter operate mode filter wakeup event and filter edge detection.

Parameters

- base – WWU peripheral base address.
- filterIndex – The index of the pin filter.
- config – Pointer to `wuu_pin_filter_config_t` structure.

```
bool WUU_GetPinFilterFlag(WUU_Type *base, uint8_t filterIndex)
```

Gets the pin filter configuration.

This function gets the pin filter flag.

Parameters

- base – WWU peripheral base address.
- filterIndex – A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

```
void WUU_ClearPinFilterFlag(WUU_Type *base, uint8_t filterIndex)
```

Clears the pin filter configuration.

This function clears the pin filter flag.

Parameters

- base – WWU peripheral base address.
- filterIndex – A pin filter index to clear the flag, starting from 1.

```
bool WUU_GetExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)
```

brief Gets the external wakeup source flag.

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

param base WWU peripheral base address. param pinIndex A pin index, which starts from 0. return True if the specific pin is a wakeup source.

```
void WUU_ClearExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)
```

brief Clears the external wakeup source flag.

This function clears the external wakeup source flag for a specific pin.

param base WWU peripheral base address. param pinIndex A pin index, which starts from 0.

FSL_WUU_DRIVER_VERSION

Defines WUU driver version 2.4.0.

enum __wuu_external_pin_edge_detection

External WakeUp pin edge detection enumeration.

Values:

enumerator kWUU_ExternalPinDisable

External input Pin disabled as wake up input.

enumerator kWUU_ExternalPinRisingEdge

External input Pin enabled with the rising edge detection.

enumerator kWUU_ExternalPinFallingEdge

External input Pin enabled with the falling edge detection.

enumerator kWUU_ExternalPinAnyEdge

External input Pin enabled with any change detection.

enum __wuu_external_wakeup_pin_event

External input wake up pin event enumeration.

Values:

enumerator kWUU_ExternalPinInterrupt

External input Pin configured as interrupt.

enumerator kWUU_ExternalPinDMARequest

External input Pin configured as DMA request.

enumerator kWUU_ExternalPinTriggerEvent

External input Pin configured as Trigger event.

enum __wuu_external_wakeup_pin_mode

External input wake up pin mode enumeration.

Values:

enumerator kWUU_ExternalPinActiveDSPD

External input Pin is active only during Deep Sleep/Power Down Mode.

enumerator kWUU_ExternalPinActiveAlways

External input Pin is active during all power modes.

enum __wuu_internal_wakeup_module_event

Internal module wake up event enumeration.

Values:

enumerator kWUU_InternalModuleInterrupt

Internal modules' interrupt as a wakeup source.

enumerator kWUU_InternalModuleDMATrigger

Internal modules' DMA/Trigger as a wakeup source.

enum __wuu_filter_edge

Pin filter edge enumeration.

Values:

enumerator kWUU_FilterDisabled

Filter disabled.

```

    enumerator kWUU_FilterPosedgeEnable
        Filter posedge detect enabled.
    enumerator kWUU_FilterNegedgeEnable
        Filter negedge detect enabled.
    enumerator kWUU_FilterAnyEdge
        Filter any edge detect enabled.
enum _wu_filter_event
    Pin Filter event enumeration.
    Values:
    enumerator kWUU_FilterInterrupt
        Filter output configured as interrupt.
    enumerator kWUU_FilterDMARequest
        Filter output configured as DMA request.
    enumerator kWUU_FilterTriggerEvent
        Filter output configured as Trigger event.
enum _wu_filter_mode
    Pin filter mode enumeration.
    Values:
    enumerator kWUU_FilterActiveDSPD
        External input pin filter is active only during Deep Sleep/Power Down Mode.
    enumerator kWUU_FilterActiveAlways
        External input Pin filter is active during all power modes.
typedef enum _wu_external_pin_edge_detection wu_external_pin_edge_detection_t
    External WakeUp pin edge detection enumeration.
typedef enum _wu_external_wakeup_pin_event wu_external_wakeup_pin_event_t
    External input wake up pin event enumeration.
typedef enum _wu_external_wakeup_pin_mode wu_external_wakeup_pin_mode_t
    External input wake up pin mode enumeration.
typedef enum _wu_internal_wakeup_module_event wu_internal_wakeup_module_event_t
    Internal module wake up event enumeration.
typedef enum _wu_filter_edge wu_filter_edge_t
    Pin filter edge enumeration.
typedef enum _wu_filter_event wu_filter_event_t
    Pin Filter event enumeration.
typedef enum _wu_filter_mode wu_filter_mode_t
    Pin filter mode enumeration.
typedef struct _wu_external_wakeup_pin_config wu_external_wakeup_pin_config_t
    External WakeUp pin configuration.
typedef struct _wu_pin_filter_config wu_pin_filter_config_t
    Pin Filter configuration.
struct _wu_external_wakeup_pin_config
    #include <fsl_wu.h> External WakeUp pin configuration.

```

Public Members

wuu_external_pin_edge_detection_t edge

External Input pin edge detection.

wuu_external_wakeup_pin_event_t event

External Input wakeup Pin event

wuu_external_wakeup_pin_mode_t mode

External Input wakeup Pin operate mode.

struct *_wuu_pin_filter_config*

#include <fsl_wuu.h> Pin Filter configuration.

Public Members

uint32_t pinIndex

The index of wakeup pin to be muxxed into filter.

wuu_filter_edge_t edge

The edge of the pin digital filter.

wuu_filter_event_t event

The event of the filter output.

wuu_filter_mode_t mode

The mode of the filter operate.

2.84 XRDC: Extended Resource Domain Controller

void XRDC_GetHardwareConfig(XRDC_Type *base, *xrdc_hardware_config_t* *config)

Gets the XRDC hardware configuration.

This function gets the XRDC hardware configurations, including number of bus masters, number of domains, number of MRCs and number of PACs.

Parameters

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

static inline void XRDC_LockGlobalControl(XRDC_Type *base)

Locks the XRDC global control register XRDC_CR.

This function locks the XRDC_CR register. After it is locked, the register is read-only until the next reset.

Parameters

- base – XRDC peripheral base address.

static inline void XRDC_SetGlobalValid(XRDC_Type *base, bool valid)

Sets the XRDC global valid.

This function sets the XRDC global valid or invalid. When the XRDC is global invalid, all accesses from all bus masters to all slaves are allowed.

Parameters

- base – XRDC peripheral base address.
- valid – True to valid XRDC.

```
static inline uint8_t XRDC_GetCurrentMasterDomainId(XRDC_Type *base)
```

Gets the domain ID of the current bus master.

This function returns the domain ID of the current bus master.

Parameters

- base – XRDC peripheral base address.

Returns

Domain ID of current bus master.

```
status_t XRDC_GetAndClearFirstDomainError(XRDC_Type *base, xrdc_error_t *error)
```

Gets and clears the first domain error of the current domain.

This function gets the first access violation information for the current domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – XRDC peripheral base address.
- error – Pointer to the error information.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_XRDC_NoError.

```
status_t XRDC_GetAndClearFirstSpecificDomainError(XRDC_Type *base, xrdc_error_t *error,
                                                    uint8_t domainId)
```

Gets and clears the first domain error of the specific domain.

This function gets the first access violation information for the specific domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – XRDC peripheral base address.
- error – Pointer to the error information.
- domainId – The error of which domain to get and clear.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_XRDC_NoError.

```
void XRDC_GetPidDefaultConfig(xrdc_pid_config_t *config)
```

Gets the default PID configuration structure.

This function initializes the configuration structure to default values. The default values are:

```
config->pid      = 0U;
config->tsmEnable = 0U;
config->sp4smEnable = 0U;
config->lockMode  = kXRDC_PidLockSecurePrivilegeWritable;
```

Parameters

- config – Pointer to the configuration structure.

```
void XRDC_SetPidConfig(XRDC_Type *base, xrdc_master_t master, const xrdc_pid_config_t
                        *config)
```

Configures the PID for a specific bus master.

This function configures the PID for a specific bus master. Do not use this function for non-processor bus masters.

Parameters

- base – XRDC peripheral base address.
- master – Which bus master to configure.
- config – Pointer to the configuration structure.

```
static inline void XRDC_SetPidLockMode(XRDC_Type *base, xrdc_master_t master,
                                       xrdc_pid_lock_t lockMode)
```

Sets the PID configuration register lock mode.

This function sets the PID configuration register lock XRDC_PIDn[LK2].

Parameters

- base – XRDC peripheral base address.
- master – Which master's PID to lock.
- lockMode – Lock mode to set.

```
void XRDC_GetDefaultNonProcessorDomainAssignment(xrdc_non_processor_domain_assignment_t
                                                  *domainAssignment)
```

Gets the default master domain assignment for non-processor bus master.

This function gets the default master domain assignment for non-processor bus master. It should only be used for the non-processor bus masters, such as DMA. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->privilegeAttr  = kXRDC_ForceUser;
assignment->privilegeAttr  = kXRDC_ForceSecure;
assignment->bypassDomainId = 0U;
assignment->blogicPartId   = 0U;
assignment->benableLogicPartId = 0U;
assignment->lock           = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

```
void XRDC_GetDefaultProcessorDomainAssignment(xrdc_processor_domain_assignment_t
                                              *domainAssignment)
```

Gets the default master domain assignment for the processor bus master.

This function gets the default master domain assignment for the processor bus master. It should only be used for the processor bus masters, such as CORE0. This function sets the assignment as follows:

```
assignment->domainId      = 0U;
assignment->domainIdSelect = kXRDC_DidMda;
assignment->dpidEnable    = kXRDC_PidDisable;
assignment->pidMask       = 0U;
assignment->pid           = 0U;
assignment->logicPartId   = 0U;
assignment->enableLogicPartId = 0U;
assignment->lock          = 0U;
```


Parameters

- domainAssignment – Pointer to the assignment structure.

```
void XRDC_SetNonProcessorDomainAssignment(XRDC_Type *base, xrdc_master_t master, uint8_t
                                         assignIndex, const
                                         xrdc_non_processor_domain_assignment_t
                                         *domainAssignment)
```

Sets the non-processor bus master domain assignment.

This function sets the non-processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for DMA0.

```
xrdc_non_processor_domain_assignment_t nonProcessorAssignment;

XRDC_GetDefaultNonProcessorDomainAssignment(&nonProcessorAssignment);
nonProcessorAssignment.domainId = 1;
nonProcessorAssignment.xxx      = xxx;

XRDC_SetMasterDomainAssignment(XRDC, kXrdcMasterDma0, 0U, &nonProcessorAssignment);
```

Parameters

- base – XRDC peripheral base address.
- master – Which master to configure.
- assignIndex – Which assignment register to set.
- domainAssignment – Pointer to the assignment structure.

```
void XRDC_SetProcessorDomainAssignment(XRDC_Type *base, xrdc_master_t master, uint8_t
                                       assignIndex, const
                                       xrdc_processor_domain_assignment_t
                                       *domainAssignment)
```

Sets the processor bus master domain assignment.

This function sets the processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to set.

Example: Set domain assignment for core 0. In this example, there are 3 assignment registers for core 0.

```
xrdc_processor_domain_assignment_t processorAssignment;

XRDC_GetDefaultProcessorDomainAssignment(&processorAssignment);

processorAssignment.domainId = 1;
processorAssignment.xxx      = xxx;
XRDC_SetMasterDomainAssignment(XRDC, kXrdcMasterCpu0, 0U, &processorAssignment);

processorAssignment.domainId = 2;
processorAssignment.xxx      = xxx;
XRDC_SetMasterDomainAssignment(XRDC, kXrdcMasterCpu0, 1U, &processorAssignment);

processorAssignment.domainId = 0;
processorAssignment.xxx      = xxx;
XRDC_SetMasterDomainAssignment(XRDC, kXrdcMasterCpu0, 2U, &processorAssignment);
```

Parameters

- base – XRDC peripheral base address.
- master – Which master to configure.
- assignIndex – Which assignment register to set.
- domainAssignment – Pointer to the assignment structure.

```
static inline void XRDC_LockMasterDomainAssignment(XRDC_Type *base, xrdc_master_t master,
                                                    uint8_t assignIndex)
```

Locks the bus master domain assignment register.

This function locks the master domain assignment. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to lock. After it is locked, the register can't be changed until next reset.

Parameters

- base – XRDC peripheral base address.
- master – Which master to configure.
- assignIndex – Which assignment register to lock.

```
static inline void XRDC_SetMasterDomainAssignmentValid(XRDC_Type *base, xrdc_master_t
                                                       master, uint8_t assignIndex, bool
                                                       valid)
```

Sets the master domain assignment as valid or invalid.

This function sets the master domain assignment as valid or invalid. One bus master might have multiple domain assignment registers. The parameter `assignIndex` specifies which assignment register to configure.

Parameters

- base – XRDC peripheral base address.
- master – Which master to configure.
- assignIndex – Index for the domain assignment register.
- valid – True to set valid, false to set invalid.

```
void XRDC_GetMemAccessDefaultConfig(xrdc_mem_access_config_t *config)
```

Gets the default memory region access policy.

This function gets the default memory region access policy. It sets the policy as follows:

```
config->enableSema      = false;
config->semaNum          = 0U;
config->subRegionDisableMask = 0U;
config->size             = kXrdcMemSizeNone;
config->lockMode         = kXRDC_AccessConfigLockWritable;
config->baseAddress      = 0U;
config->policy[0]        = kXRDC_AccessPolicyNone;
config->policy[1]        = kXRDC_AccessPolicyNone;
...
config->policy[15]       = kXRDC_AccessPolicyNone;
```

Parameters

- config – Pointer to the configuration structure.

```
void XRDC_SetMemAccessConfig(XRDC_Type *base, const xrdc_mem_access_config_t *config)
```

Sets the memory region access policy.

This function sets the memory region access configuration as valid. There are two methods to use it:

Example 1: Set one configuration run time. Set memory region 0x20000000 ~ 0x20000400 accessible by domain 0, use MRC0_1.

```
xrdc_mem_access_config_t config =
{
    .mem      = kXRDC_MemMrc0_1,
    .baseAddress = 0x20000000U,
    .size     = kXRDC_MemSize1K,
    .policy[0] = kXRDC_AccessPolicyAll
};
XRDC_SetMemAccessConfig(XRDC, &config);
```

Example 2: Set multiple configurations during startup. Set memory region 0x20000000 ~ 0x20000400 accessible by domain 0, use MRC0_1. Set memory region 0x1FFF0000 ~ 0x1FFF0800 accessible by domain 0, use MRC0_2.

```
xrdc_mem_access_config_t configs[] =
{
    {
        .mem      = kXRDC_MemMrc0_1,
        .baseAddress = 0x20000000U,
        .size     = kXRDC_MemSize1K,
        .policy[0] = kXRDC_AccessPolicyAll
    },
    {
        .mem      = kXRDC_MemMrc0_2,
        .baseAddress = 0x1FFF0000U,
        .size     = kXRDC_MemSize2K,
        .policy[0] = kXRDC_AccessPolicyAll
    }
};

for (i=0U; i<((sizeof(configs)/sizeof(configs[0]))); i++)
{
    XRDC_SetMemAccessConfig(XRDC, &configs[i]);
}
```

Parameters

- base – XRDC peripheral base address.
- config – Pointer to the access policy configuration structure.

```
static inline void XRDC_SetMemAccessLockMode(XRDC_Type *base, xrdc_mem_t mem,
                                             xrdc_access_config_lock_t lockMode)
```

Sets the memory region descriptor register lock mode.

Parameters

- base – XRDC peripheral base address.
- mem – Which memory region descriptor to lock.
- lockMode – The lock mode to set.

```
static inline void XRDC_SetMemAccessValid(XRDC_Type *base, xrdc_mem_t mem, bool valid)
```

Sets the memory region descriptor as valid or invalid.

This function sets the memory region access configuration dynamically. For example:

```
xrdc_mem_access_config_t config =
{
    .mem      = kXRDC_MemMrc0_1,
    .baseAddress = 0x20000000U,
```

(continues on next page)

(continued from previous page)

```

.size      = kXRDC_MemSize1K,
.policy[0] = kXRDC_AccessPolicyAll
};
XRDC_SetMemAccessConfig(XRDC, &config);

XRDC_SetMemAccessValid(kXRDC_MemMrc0_1, false);

XRDC_SetMemAccessValid(kXRDC_MemMrc0_1, true);

```

Parameters

- base – XRDC peripheral base address.
- mem – Which memory region descriptor to set.
- valid – True to set valid, false to set invalid.

void XRDC_GetPeriphAccessDefaultConfig(*xrdc_periph_access_config_t* *config)

Gets the default peripheral access configuration.

The default configuration is set as follows:

```

config->enableSema    = false;
config->semaNum       = 0U;
config->lockMode      = kXRDC_AccessConfigLockWritable;
config->policy[0]     = kXRDC_AccessPolicyNone;
config->policy[1]     = kXRDC_AccessPolicyNone;
...
config->policy[15]    = kXRDC_AccessPolicyNone;

```

Parameters

- config – Pointer to the configuration structure.

void XRDC_SetPeriphAccessConfig(XRDC_Type *base, const *xrdc_periph_access_config_t* *config)

Sets the peripheral access configuration.

This function sets the peripheral access configuration as valid. Two methods to use it:
Method 1: Set for one peripheral, which is used for runtime settings. Example: set LPTMR0 accessible by domain 0

```

xrdc_periph_access_config_t config;

config.periph = kXRDC_PeriphLptmr0;
config.policy[0] = kXRDC_AccessPolicyAll;
XRDC_SetPeriphAccessConfig(XRDC, &config);

```

Method 2: Set for multiple peripherals, which is used for initialization settings.

```

xrdc_periph_access_config_t configs[] =
{
    {
        .periph = kXRDC_PeriphLptmr0,
        .policy[0] = kXRDC_AccessPolicyAll,
        .policy[1] = kXRDC_AccessPolicyAll
    },
    {
        .periph = kXRDC_PeriphLpuart0,
        .policy[0] = kXRDC_AccessPolicyAll,
        .policy[1] = kXRDC_AccessPolicyAll
    }
};

```

(continues on next page)

(continued from previous page)

```

for (i=0U; i<(sizeof(configs)/sizeof(configs[0])), i++)
{
    XRDC_SetPeriphAccessConfig(XRDC, &config[i]);
}

```

Parameters

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

```
static inline void XRDC_SetPeriphAccessLockMode(XRDC_Type *base, xrdc_periph_t periph,
                                                xrdc_access_config_lock_t lockMode)
```

Sets the peripheral access configuration register lock mode.

Parameters

- base – XRDC peripheral base address.
- periph – Which peripheral access configuration register to lock.
- lockMode – The lock mode to set.

```
static inline void XRDC_SetPeriphAccessValid(XRDC_Type *base, xrdc_periph_t periph, bool
                                              valid)
```

Sets the peripheral access as valid or invalid.

This function sets the peripheral access configuration dynamically. For example:

```

xrdc_periph_access_config_t config =
{
    .periph    = kXRDC_PeriphLptmr0;
    .policy[0] = kXRDC_AccessPolicyAll;
};
XRDC_SetPeriphAccessConfig(XRDC, &config);

XRDC_SetPeriphAccessValid(kXrdcPeriLptmr0, false);

XRDC_SetPeriphAccessValid(kXrdcPeriLptmr0, true);

```

Parameters

- base – XRDC peripheral base address.
- periph – Which peripheral access configuration to set.
- valid – True to set valid, false to set invalid.

XRDC status `_xrdc_status`.

Values:

enumerator `kStatus_XRDC_NoError`

No error captured.

enum `_xrdc_pid_enable`

XRDC PID enable mode, the register bit `XRDC_MDA_Wx[PE]`, used for domain hit evaluation.

Values:

enumerator `kXRDC_PidDisable`

PID is not used in domain hit evaluation.

enumerator kXRDC_PidDisable1

PID is not used in domain hit evaluation.

enumerator kXRDC_PidExp0

$((\text{XRDC\_MDA\_W}[\text{PID}] \ \& \ \sim \text{XRDC\_MDA\_W}[\text{PIDM}]) \ == \ (\text{XRDC\_PID}[\text{PID}] \ \& \ \sim \text{XRDC\_MDA\_W}[\text{PIDM}]))$.

enumerator kXRDC_PidExp1

$(\sim(\text{XRDC\_MDA\_W}[\text{PID}] \ \& \ \sim \text{XRDC\_MDA\_W}[\text{PIDM}]) \ == \ (\text{XRDC\_PID}[\text{PID}] \ \& \ \sim \text{XRDC\_MDA\_W}[\text{PIDM}]))$.

enum _xrdc_did_sel

XRDC domain ID select method, the register bit XRDC_MDA_Wx[DIDS], used for domain hit evaluation.

Values:

enumerator kXRDC_DidMda

Use MDAn[3:0] as DID.

enumerator kXRDC_DidInput

Use the input DID (DID_in) as DID.

enumerator kXRDC_DidMdaAndInput

Use MDAn[3:2] concatenated with DID_in[1:0] as DID.

enumerator kXRDC_DidReserved

Reserved.

enum _xrdc_secure_attr

XRDC secure attribute, the register bit XRDC_MDA_Wx[SA], used for non-processor bus master domain assignment.

Values:

enumerator kXRDC_ForceSecure

Force the bus attribute for this master to secure.

enumerator kXRDC_ForceNonSecure

Force the bus attribute for this master to non-secure.

enumerator kXRDC_MasterSecure

Use the bus master's secure/nonsecure attribute directly.

enumerator kXRDC_MasterSecure1

Use the bus master's secure/nonsecure attribute directly.

enum _xrdc_privilege_attr

XRDC privileged attribute, the register bit XRDC_MDA_Wx[PA], used for non-processor bus master domain assignment.

Values:

enumerator kXRDC_ForceUser

Force the bus attribute for this master to user.

enumerator kXRDC_ForcePrivilege

Force the bus attribute for this master to privileged.

enumerator kXRDC_MasterPrivilege

Use the bus master's attribute directly.

enumerator kXRDC_MasterPrivilege1

Use the bus master's attribute directly.

enum _xrdc_pid_lock

XRDC PID LK2 definition XRDC_PIDn[LK2].

Values:

enumerator kXRDC_PidLockSecurePrivilegeWritable

Writable by any secure privileged write.

enumerator kXRDC_PidLockSecurePrivilegeWritable1

Writable by any secure privileged write.

enumerator kXRDC_PidLockMasterXOnly

PIDx is only writable by master x.

enumerator kXRDC_PidLockLocked

Read-only until the next reset.

enum _xrdc_access_policy

XRDC domain access control policy.

Values:

enumerator kXRDC_AccessPolicyNone

enumerator kXRDC_AccessPolicySpuR

enumerator kXRDC_AccessPolicySpRw

enumerator kXRDC_AccessPolicySpuRw

enumerator kXRDC_AccessPolicySpuRwNpR

enumerator kXRDC_AccessPolicySpuRwNpuR

enumerator kXRDC_AccessPolicySpuRwNpRw

enumerator kXRDC_AccessPolicyAll

enum _xrdc_access_config_lock

Access configuration lock mode, the register field PDAC and MRGD LK2.

Values:

enumerator kXRDC_AccessConfigLockWritable

Entire PDACn/MRGDn can be written.

enumerator kXRDC_AccessConfigLockWritable1

Entire PDACn/MRGDn can be written.

enumerator kXRDC_AccessConfigLockDomainXOnly

Domain x only write the DxACP field.

enumerator kXRDC_AccessConfigLockLocked

PDACn is read-only until the next reset.

enum _xrdc_mem_size

XRDC memory size definition.

Values:

enumerator kXRDC_MemSizeNone

None size.

enumerator kXRDC_MemSize32B

$2^{(4+1)} = 32$

enumerator kXRDC_MemSize64B
 $2^{(5+1)} = 64$

enumerator kXRDC_MemSize128B
 $2^{(6+1)} = 128$

enumerator kXRDC_MemSize256B
 $2^{(7+1)} = 256$

enumerator kXRDC_MemSize512B
 $2^{(8+1)} = 512$

enumerator kXRDC_MemSize1K
 $2^{(9+1)} = 1\text{kB}$

enumerator kXRDC_MemSize2K
 $2^{(10+1)} = 2\text{kB}$

enumerator kXRDC_MemSize4K
 $2^{(11+1)} = 4\text{kB}$

enumerator kXRDC_MemSize8K
 $2^{(12+1)} = 8\text{kB}$

enumerator kXRDC_MemSize16K
 $2^{(13+1)} = 16\text{kB}$

enumerator kXRDC_MemSize32K
 $2^{(14+1)} = 32\text{kB}$

enumerator kXRDC_MemSize64K
 $2^{(15+1)} = 64\text{kB}$

enumerator kXRDC_MemSize128K
 $2^{(16+1)} = 128\text{kB}$

enumerator kXRDC_MemSize256K
 $2^{(17+1)} = 256\text{kB}$

enumerator kXRDC_MemSize512K
 $2^{(18+1)} = 512\text{kB}$

enumerator kXRDC_MemSize1M
 $2^{(19+1)} = 1\text{MB}$

enumerator kXRDC_MemSize2M
 $2^{(20+1)} = 2\text{MB}$

enumerator kXRDC_MemSize4M
 $2^{(21+1)} = 4\text{MB}$

enumerator kXRDC_MemSize8M
 $2^{(22+1)} = 8\text{MB}$

enumerator kXRDC_MemSize16M
 $2^{(23+1)} = 16\text{MB}$

enumerator kXRDC_MemSize32M
 $2^{(24+1)} = 32\text{MB}$

enumerator kXRDC_MemSize64M
 $2^{(25+1)} = 64\text{MB}$

enumerator kXRDC_MemSize128M

$2^{(26+1)} = 128\text{MB}$

enumerator kXRDC_MemSize256M

$2^{(27+1)} = 256\text{MB}$

enumerator kXRDC_MemSize512M

$2^{(28+1)} = 512\text{MB}$

enumerator kXRDC_MemSize1G

$2^{(29+1)} = 1\text{GB}$

enumerator kXRDC_MemSize2G

$2^{(30+1)} = 2\text{GB}$

enumerator kXRDC_MemSize4G

$2^{(31+1)} = 4\text{GB}$

enum _xrdc_controller

XRDC controller definition for domain error check.

Values:

enumerator kXRDC_MemController0

Memory region controller 0.

enumerator kXRDC_MemController1

Memory region controller 1.

enumerator kXRDC_MemController2

Memory region controller 2.

enumerator kXRDC_MemController3

Memory region controller 3.

enumerator kXRDC_MemController4

Memory region controller 4.

enumerator kXRDC_MemController5

Memory region controller 5.

enumerator kXRDC_MemController6

Memory region controller 6.

enumerator kXRDC_MemController7

Memory region controller 7.

enumerator kXRDC_MemController8

Memory region controller 8.

enumerator kXRDC_MemController9

Memory region controller 9.

enumerator kXRDC_MemController10

Memory region controller 10.

enumerator kXRDC_MemController11

Memory region controller 11.

enumerator kXRDC_MemController12

Memory region controller 12.

enumerator kXRDC_MemController13

Memory region controller 13.

enumerator kXRDC_MemController14

Memory region controller 14.

enumerator kXRDC_MemController15

Memory region controller 15.

enumerator kXRDC_PeriphController0

Peripheral access controller 0.

enumerator kXRDC_PeriphController1

Peripheral access controller 1.

enumerator kXRDC_PeriphController2

Peripheral access controller 2.

enumerator kXRDC_PeriphController3

Peripheral access controller 3.

enum _xrdc_error_state

XRDC domain error state definition XRDC_DERR_W1_n[EST].

Values:

enumerator kXRDC_ErrorStateNone

No access violation detected.

enumerator kXRDC_ErrorStateNone1

No access violation detected.

enumerator kXRDC_ErrorStateSingle

Single access violation detected.

enumerator kXRDC_ErrorStateMulti

Multiple access violation detected.

enum _xrdc_error_attr

XRDC domain error attribute definition XRDC_DERR_W1_n[EATR].

Values:

enumerator kXRDC_ErrorSecureUserInst

Secure user mode, instruction fetch access.

enumerator kXRDC_ErrorSecureUserData

Secure user mode, data access.

enumerator kXRDC_ErrorSecurePrivilegeInst

Secure privileged mode, instruction fetch access.

enumerator kXRDC_ErrorSecurePrivilegeData

Secure privileged mode, data access.

enumerator kXRDC_ErrorNonSecureUserInst

NonSecure user mode, instruction fetch access.

enumerator kXRDC_ErrorNonSecureUserData

NonSecure user mode, data access.

enumerator kXRDC_ErrorNonSecurePrivilegeInst

NonSecure privileged mode, instruction fetch access.

enumerator kXRDC_ErrorNonSecurePrivilegeData
NonSecure privileged mode, data access.

enum _xrdc_error_type
XRDC domain error access type definition XRDC_DERR_W1_n[ERW].

Values:

enumerator kXRDC_ErrorTypeRead
Error occurs on read reference.

enumerator kXRDC_ErrorTypeWrite
Error occurs on write reference.

typedef struct _xrdc_hardware_config xrdc_hardware_config_t
XRDC hardware configuration.

typedef enum _xrdc_pid_enable xrdc_pid_enable_t
XRDC PID enable mode, the register bit XRDC_MDA_Wx[PE], used for domain hit evaluation.

typedef enum _xrdc_did_sel xrdc_did_sel_t
XRDC domain ID select method, the register bit XRDC_MDA_Wx[DIDS], used for domain hit evaluation.

typedef enum _xrdc_secure_attr xrdc_secure_attr_t
XRDC secure attribute, the register bit XRDC_MDA_Wx[SA], used for non-processor bus master domain assignment.

typedef enum _xrdc_privilege_attr xrdc_privilege_attr_t
XRDC privileged attribute, the register bit XRDC_MDA_Wx[PA], used for non-processor bus master domain assignment.

typedef struct _xrdc_processor_domain_assignment xrdc_processor_domain_assignment_t
Domain assignment for the processor bus master.

typedef struct _xrdc_non_processor_domain_assignment
xrdc_non_processor_domain_assignment_t
Domain assignment for the non-processor bus master.

typedef enum _xrdc_pid_lock xrdc_pid_lock_t
XRDC PID LK2 definition XRDC_PIDn[LK2].

typedef struct _xrdc_pid_config xrdc_pid_config_t
XRDC process identifier (PID) configuration.

typedef enum _xrdc_access_policy xrdc_access_policy_t
XRDC domain access control policy.

typedef enum _xrdc_access_config_lock xrdc_access_config_lock_t
Access configuration lock mode, the register field PDAC and MRGD LK2.

typedef struct _xrdc_periph_access_config xrdc_periph_access_config_t
XRDC peripheral domain access control configuration.

typedef enum _xrdc_mem_size xrdc_mem_size_t
XRDC memory size definition.

typedef struct _xrdc_mem_access_config xrdc_mem_access_config_t
XRDC memory region domain access control configuration.

typedef enum _xrdc_controller xrdc_controller_t
XRDC controller definition for domain error check.

```
typedef enum _xrdc_error_state xrdc_error_state_t
```

XRDC domain error state definition XRDC_DERR_W1_n[EST].

```
typedef enum _xrdc_error_attr xrdc_error_attr_t
```

XRDC domain error attribute definition XRDC_DERR_W1_n[EATR].

```
typedef enum _xrdc_error_type xrdc_error_type_t
```

XRDC domain error access type definition XRDC_DERR_W1_n[ERW].

```
typedef struct _xrdc_error xrdc_error_t
```

XRDC domain error definition.

```
void XRDC_Init(XRDC_Type *base)
```

Initializes the XRDC module.

This function enables the XRDC clock.

Parameters

- base – XRDC peripheral base address.

```
void XRDC_Deinit(XRDC_Type *base)
```

De-initializes the XRDC module.

This function disables the XRDC clock.

Parameters

- base – XRDC peripheral base address.

```
FSL_XRDC_DRIVER_VERSION
```

```
struct _xrdc_hardware_config
```

#include <fsl_xrdc.h> XRDC hardware configuration.

Public Members

```
uint8_t masterNumber
```

Number of bus masters.

```
uint8_t domainNumber
```

Number of domains.

```
uint8_t pacNumber
```

Number of PACs.

```
uint8_t mrcNumber
```

Number of MRCs.

```
struct _xrdc_processor_domain_assignment
```

#include <fsl_xrdc.h> Domain assignment for the processor bus master.

Public Members

```
uint32_t domainId
```

Domain ID.

```
uint32_t domainIdSelect
```

Domain ID select method, see `xrdc_did_sel_t`.

```
uint32_t pidEnable
```

PId enable method, see `xrdc_pid_enable_t`.

uint32_t pidMask
Pid mask.

uint32_t __pad0__
Reserved.

uint32_t pid
Pid value.

uint32_t __pad1__
Reserved.

uint32_t logicPartId
Logical partition ID.

uint32_t enableLogicPartId
Logical partition ID.

uint32_t __pad2__
Reserved.

uint32_t lock
Lock the register.

uint32_t __pad3__
Reserved.

struct __xrdc_non_processor_domain_assignment
#include <fsl_xrdc.h> Domain assignment for the non-processor bus master.

Public Members

uint32_t domainId
Domain ID.

uint32_t privilegeAttr
Privileged attribute, see `xrdc_privilege_attr_t`.

uint32_t secureAttr
Secure attribute, see `xrdc_secure_attr_t`.

uint32_t bypassDomainId
Bypass domain ID.

uint32_t __pad0__
Reserved.

uint32_t logicPartId
Logical partition ID.

uint32_t enableLogicPartId
Enable logical partition ID.

uint32_t __pad1__
Reserved.

uint32_t lock
Lock the register.

uint32_t __pad2__
Reserved.

struct __xrdc_pid_config
#include <fsl_xrdc.h> XRDC process identifier (PID) configuration.

Public Members

uint32_t pid

PID value, PIDn[PID].

uint32_t __pad0__

Reserved.

uint32_t tsmEnable

Enable three-state model.

uint32_t lockMode

PIDn configuration lock mode, see `xrdc_pid_lock_t`.

uint32_t __pad1__

Reserved.

struct `_xrdc_periph_access_config`

#include <fsl_xrdc.h> XRDC peripheral domain access control configuration.

Public Members

xrdc_periph_t periph

Peripheral name.

xrdc_access_config_lock_t lockMode

PDACn lock configuration.

bool enableSema

Enable semaphore or not.

uint32_t semaNum

Semaphore number.

xrdc_access_policy_t policy[FSL_FEATURE_XRDC_DOMAIN_COUNT]

Access policy for each domain.

struct `_xrdc_mem_access_config`

#include <fsl_xrdc.h> XRDC memory region domain access control configuration.

Public Members

xrdc_mem_t mem

Memory region descriptor name.

bool enableSema

Enable semaphore or not.

uint8_t semaNum

Semaphore number.

xrdc_mem_size_t size

Memory region size.

uint8_t subRegionDisableMask

Sub-region disable mask.

xrdc_access_config_lock_t lockMode

MRGDn lock configuration.

xrdc_access_policy_t policy[FSL_FEATURE_XRDC_DOMAIN_COUNT]

Access policy for each domain.

uint32_t baseAddress

Memory region base/start address.

struct __xrdc_error

#include <fsl_xrdc.h> XRDC domain error definition.

Public Members

xrdc_controller_t controller

Which controller captured access violation.

uint32_t address

Access address that generated access violation.

xrdc_error_state_t errorState

Error state.

xrdc_error_attr_t errorAttr

Error attribute.

xrdc_error_type_t errorType

Error type.

uint8_t errorPort

Error port.

uint8_t domainId

Domain ID.

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.

3.2 MultiCore

3.2.1 Multicore SDK

Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

Multicore SDK (MCSDK) Release Notes

Overview These are the release notes for the NXP Multicore Software Development Kit (MCSDK) version 25.06.00.

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

What is new

- eRPC [CHANGELOG](#)
- RPPMsg-Lite [CHANGELOG](#)
- MCMgr [CHANGELOG](#)
- Supported evaluation boards (multicore examples):
 - LPCXpresso55S69
 - FRDM-K32L3A6
 - MIMXRT1170-EVKB
 - MIMXRT1160-EVK
 - MIMXRT1180-EVK
 - MCX-N5XX-EVK
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK
 - KW47-EVK
 - KW47-LOC
 - FRDM-MCXW72
 - MCX-W72-EVK
- Supported evaluation boards (multiprocessor examples):
 - LPCXpresso55S36
 - FRDM-K22F
 - FRDM-K32L2B
 - MIMXRT685-EVK
 - MIMXRT1170-EVKB
 - MIMXRT1180
 - FRDM-MCXN236
 - FRDM-MCXC242
 - FRDM-MCXC444
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK

Development tools The Multicore SDK (MCSDK) was compiled and tested with development tools referred in: [Development tools](#)

Release contents This table describes the release contents. Not all MCUXpresso SDK packages contain the whole set of these components.

Deliverable	Location
Multicore SDK location	<MCUXpressoSDK_install_dir>/middleware/multicore/
Documentation	<MCSDK_dir>/mcuxsdk-doc/
Embedded Remote Procedure Call component	<MCSDK_dir>/erpc/
Multicore Manager component	<MCSDK_dir>/mcmgr/
RPMMsg-Lite	<MCSDK_dir>/rpmsg_lite/
Multicore demo applications	<MCUXpressoSDK_install_dir>/examples/multicore_examples/
Multiprocessor demo applications	<MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/

Multicore SDK release overview Together, the Multicore SDK (MCSDK) and the MCUXpresso SDK (SDK) form a framework for the development of software for NXP multicore devices. The MCSDK release consists of the following elementary software components for multicore:

- Embedded Remote Procedure Call (eRPC)
- Multicore Manager (MCMGR) - included just in SDK for multicore devices
- Remote Processor Messaging - Lite (RPMMsg-Lite) - included just in SDK for multicore devices

The MCSDK is also accompanied with documentation and several multicore and multiprocessor demo applications.

Demo applications The multicore demo applications demonstrate the usage of the MCSDK software components on supported multicore development boards.

The following multicore demo applications are located together with other MCUXpresso SDK examples in

the <MCUXpressoSDK_install_dir>/examples/multicore_examples subdirectories.

- erpc_matrix_multiply_mu
- erpc_matrix_multiply_mu_rtos
- erpc_matrix_multiply_rpmsg
- erpc_matrix_multiply_rpmsg_rtos
- erpc_two_way_rpc_rpmsg_rtos
- freertos_message_buffers
- hello_world
- multicore_manager
- rpmsg_lite_pingpong
- rpmsg_lite_pingpong_rtos
- rpmsg_lite_pingpong_tzm

The eRPC multicore component can be leveraged for inter-processor communication and remote procedure calls between SoCs / development boards.

The following multiprocessor demo applications are located together with other MCUXpresso SDK examples in

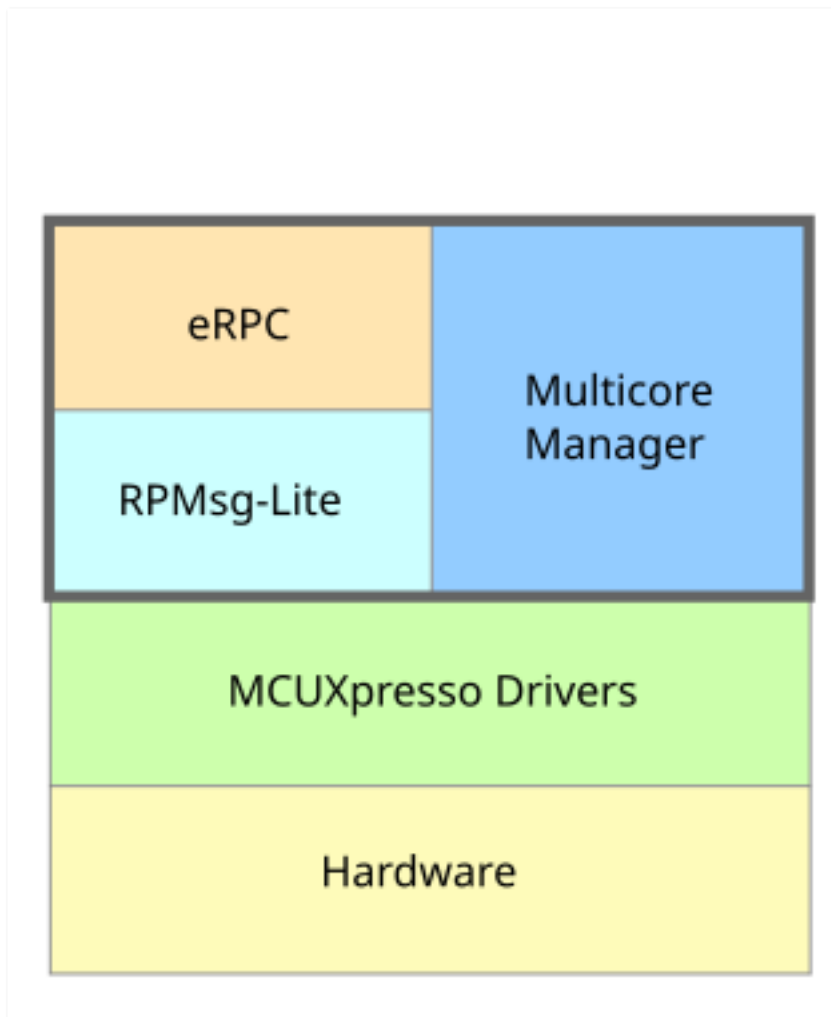
the <MCUXpressoSDK_install_dir>/examples/multiprocessor_examples subdirectories.

- erpc_client_matrix_multiply_spi
- erpc_server_matrix_multiply_spi
- erpc_client_matrix_multiply_uart
- erpc_server_matrix_multiply_uart
- erpc_server_dac_adc
- erpc_remote_control

Getting Started with Multicore SDK (MCSDK)

Overview Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

The following figure highlights the layers and main software components of the MCSDK.

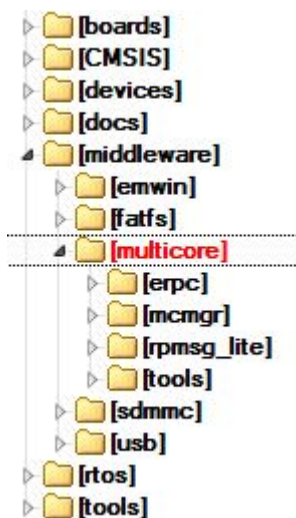


All the MCSDK-related files are located in `<MCUXpressoSDK_install_dir>/middleware/multicore` folder.

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

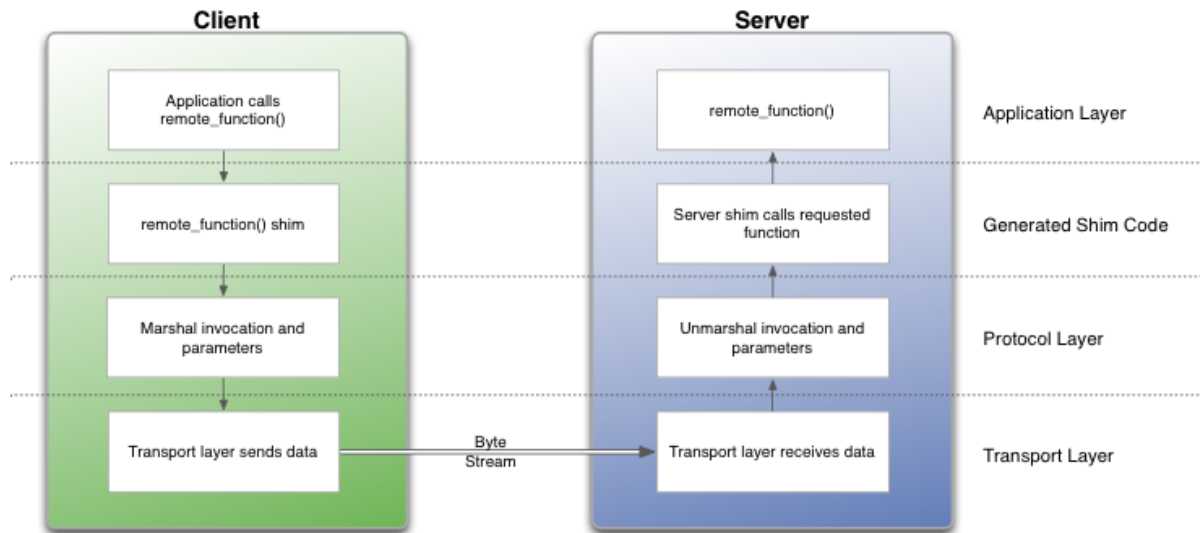
Multicore SDK (MCSDK) components The MCSDK consists of the following software components:

- **Embedded Remote Procedure Call (eRPC):** This component is a combination of a library and code generator tool that implements a transparent function call interface to remote services (running on a different core).
- **Multicore Manager (MCMGR):** This library maintains information about all cores and starts up secondary/auxiliary cores.
- **Remote Processor Messaging - Lite (RPMsg-Lite):** Inter-Processor Communication library.



Embedded Remote Procedure Call (eRPC) The Embedded Remote Procedure Call (eRPC) is the RPC system created by NXP. The RPC is a mechanism used to invoke a software routine on a remote system via a simple local function call.

When a remote function is called by the client, the function's parameters and an identifier for the called routine are marshaled (or serialized) into a stream of bytes. This byte stream is transported to the server through a communications channel (IPC, TPC/IP, UART, and so on). The server unmarshals the parameters, determines which function was invoked, and calls it. If the function returns a value, it is marshaled and sent back to the client.



RPC implementations typically use a combination of a tool (erpcgen) and IDL (interface definition language) file to generate source code to handle the details of marshaling a function's parameters and building the data stream.

Main eRPC features:

- Scalable from BareMetal to Linux OS - configurable memory and threading policies.
- Focus on embedded systems - intrinsic support for C, modular, and lightweight implementation.
- Abstracted transport interface - RPMsg is the primary transport for multicore, UART, or SPI-based solutions can be used for multichip.

The eRPC library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc` folder. For detailed information about the eRPC, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems.

The main MCMGR features:

- Maintains information about all cores in system.
- Secondary/auxiliary cores startup and shutdown.
- Remote core monitoring and event handling.

The MCMGR library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr` folder. For detailed information about the MCMGR library, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/doc` folder.

Remote Processor Messaging Lite (RPMsg-Lite) RPMsg-Lite is a lightweight implementation of the RPMsg protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. Compared to the legacy OpenAMP implementation, RPMsg-Lite offers a code size reduction, API simplification, and improved modularity.

The main RPMsg protocol features:

- Shared memory interprocessor communication.
- Virtio-based messaging bus.
- Application-defined messages sent between endpoints.

- Portable to different environments/platforms.
- Available in upstream Linux OS.

The RPMsg-Lite library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg-lite` folder. For detailed information about the RPMsg-Lite, see the RPMsg-Lite User's Guide located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/doc` folder.

MCSDK demo applications Multicore and multiprocessor example applications are stored together with other MCUXpresso SDK examples, in the dedicated multicore subfolder.

Location		Folder
Multicore projects	example	<code><MCUXpressoSDK_install_dir>/examples/multicore_examples/<application_name>/</code>
Multiprocessor projects	example	<code><MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/<application_name>/</code>

See the *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) and *Getting Started with MCUXpresso SDK for XXX Derivatives* documents for more information about the MCUXpresso SDK example folder structure and the location of individual files that form the example application projects. These documents also contain information about building, running, and debugging multicore demo applications in individual supported IDEs. Each example application also contains a readme file that describes the operation of the example and required setup steps.

Inter-Processor Communication (IPC) levels The MCSDK provides several mechanisms for Inter-Processor Communication (IPC). Particular ways and levels of IPC are described in this chapter.

IPC using low-level drivers

The NXP multicore SoCs are equipped with peripheral modules dedicated for data exchange between individual cores. They deal with the Mailbox peripheral for LPC parts and the Messaging Unit (MU) peripheral for Kinetis and i.MX parts. The common attribute of both modules is the ability to provide a means of IPC, allowing multiple CPUs to share resources and communicate with each other in a simple manner.

The most lightweight method of IPC uses the MCUXpresso SDK low-level drivers for these peripherals. Using the Mailbox/MU driver API functions, it is possible to pass a value from core to core via the dedicated registers (could be a scalar or a pointer to shared memory) and also to trigger inter-core interrupts for notifications.

For details about individual driver API functions, see the MCUXpresso SDK API Reference Manual of the specific multicore device. The MCUXpresso SDK is accompanied with the RPMsg-Lite documentation that shows how to use this API in multicore applications.

Messaging mechanism

On top of Mailbox/MU drivers, a messaging system can be implemented, allowing messages to send between multiple endpoints created on each of the CPUs. The RPMsg-Lite library of the MCSDK provides this ability and serves as the preferred MCUXpresso SDK messaging library. It implements ring buffers in shared memory for messages exchange without the need of a locking mechanism.

The RPMsg-Lite provides the abstraction layer and can be easily ported to different multicore platforms and environments (Operating Systems). The advantages of such a messaging system are ease of use (there is no need to study behavior of the used underlying hardware) and smooth application code portability between platforms due to unified messaging API.

However, this costs several kB of code and data memory. The MCUXpresso SDK is accompanied by the RPMsg-Lite documentation and several multicore examples. You can also obtain the latest RPMsg-Lite code from the GitHub account github.com/nxp-mcuxpresso/rpmsg-lite.

Remote procedure calls

To facilitate the IPC even more and to allow the remote functions invocation, the remote procedure call mechanism can be implemented. The eRPC of the MCSDK serves for these purposes and allows the ability to invoke a software routine on a remote system via a simple local function call. Utilizing different transport layers, it is possible to communicate between individual cores of multicore SoCs (via RPMsg-Lite) or between separate processors (via SPI, UART, or TCP/IP). The eRPC is mostly applicable to the MPU parts with enough of memory resources like i.MX parts.

The eRPC library allows you to export existing C functions without having to change their prototypes (in most cases). It is accompanied by the code generator tool that generates the shim code for serialization and invocation based on the IDL file with definitions of data types and remote interfaces (API).

If the communicating peer is running as a Linux OS user-space application, the generated code can be either in C/C++ or Python.

Using the eRPC simplifies the access to services implemented on individual cores. This way, the following types of applications running on dedicated cores can be easily interfaced:

- Communication stacks (USB, Thread, Bluetooth Low Energy, Zigbee)
- Sensor aggregation/fusion applications
- Encryption algorithms
- Virtual peripherals

The eRPC is publicly available from the following GitHub account: github.com/EmbeddedRPC/erpc. Also, the MCUXpresso SDK is accompanied by the eRPC code and several multicore and multiprocessor eRPC examples.

The mentioned IPC levels demonstrate the scalability of the Multicore SDK library. Based on application needs, different IPC techniques can be used. It depends on the complexity, required speed, memory resources, system design, and so on. The MCSDK brings users the possibility for quick and easy development of multicore and multiprocessor applications.

Changelog Multicore SDK

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[25.06.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.14.0
 - eRPC generator (erpcgen) v1.14.0
 - Multicore Manager (MCMgr) v5.0.0
 - RPMsg-Lite v5.2.0

[25.03.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0

- eRPC generator (erpcgen) v1.13.0
- Multicore Manager (MCMgr) v4.1.7
- RPPMsg-Lite v5.1.4

[24.12.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.6
 - RPPMsg-Lite v5.1.3

[2.16.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.2

[2.15.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.12.0
 - eRPC generator (erpcgen) v1.12.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.1

[2.14.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.11.0
 - eRPC generator (erpcgen) v1.11.0
 - Multicore Manager (MCMgr) v4.1.4
 - RPPMsg-Lite v5.1.0

[2.13.0_imxrt1180a0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPPMsg-Lite v5.0.0

[2.13.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPSMsg-Lite v5.0.0

[2.12.0_imx93]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.1

[2.12.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.0

[2.11.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.1

[2.11.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.0

[2.10.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.1
 - eRPC generator (erpcgen) v1.8.1
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.2

[2.9.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.0
 - eRPC generator (erpcgen) v1.8.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.1

[2.8.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.4
 - eRPC generator (erpcgen) v1.7.4
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.1.0

[2.7.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.3
 - eRPC generator (erpcgen) v1.7.3
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.0.0

[2.6.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.2
 - eRPC generator (erpcgen) v1.7.2
 - Multicore Manager (MCMgr) v4.0.3
 - RPSMsg-Lite v2.2.0

[2.5.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.1
 - eRPC generator (erpcgen) v1.7.1
 - Multicore Manager (MCMgr) v4.0.2
 - RPMsg-Lite v2.0.2

[2.4.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.0
 - eRPC generator (erpcgen) v1.7.0
 - Multicore Manager (MCMgr) v4.0.1
 - RPMsg-Lite v2.0.1

[2.3.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.6.0
 - eRPC generator (erpcgen) v1.6.0
 - Multicore Manager (MCMgr) v4.0.0
 - RPMsg-Lite v1.2.0

[2.3.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.5.0
 - eRPC generator (erpcgen) v1.5.0
 - Multicore Manager (MCMgr) v3.0.0
 - RPMsg-Lite v1.2.0

[2.2.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.4.0
 - eRPC generator (erpcgen) v1.4.0
 - Multicore Manager (MCMgr) v2.0.1
 - RPMsg-Lite v1.1.0

[2.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.3.0
 - eRPC generator (erpcgen) v1.3.0

[2.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.2.0
 - eRPC generator (erpcgen) v1.2.0
 - Multicore Manager (MCMgr) v2.0.0
 - RPMMsg-Lite v1.0.0

[1.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.1.0
 - Multicore Manager (MCMgr) v1.1.0
 - Open-AMP / RPMMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev01

[1.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.0.0
 - Multicore Manager (MCMgr) v1.0.0
 - Open-AMP / RPMMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev00

Multicore SDK Components

RPMSG-Lite

MCUXpresso SDK : mcuxsdk-middleware-rpmsg-lite

Overview This repository is for MCUXpresso SDK RPMSG-Lite middleware delivery and it contains RPMSG-Lite component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run RPMSG-Lite examples that are based on mcux-sdk-middleware-rpmsg-lite component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

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

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the rpmsg-lite project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

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

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

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

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

For overview please read RPMSG-Lite VirtIO Overview.

For RPMSG-Lite Design Considerations please read RPMSG-Lite Design Considerations.

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

RPMsg protocol was not documented, and its only definition was given by the Linux Kernel and legacy OpenAMP implementations. This has changed with [1] which is a standardization protocol allowing multiple different implementations to coexist and still be mutually compatible.

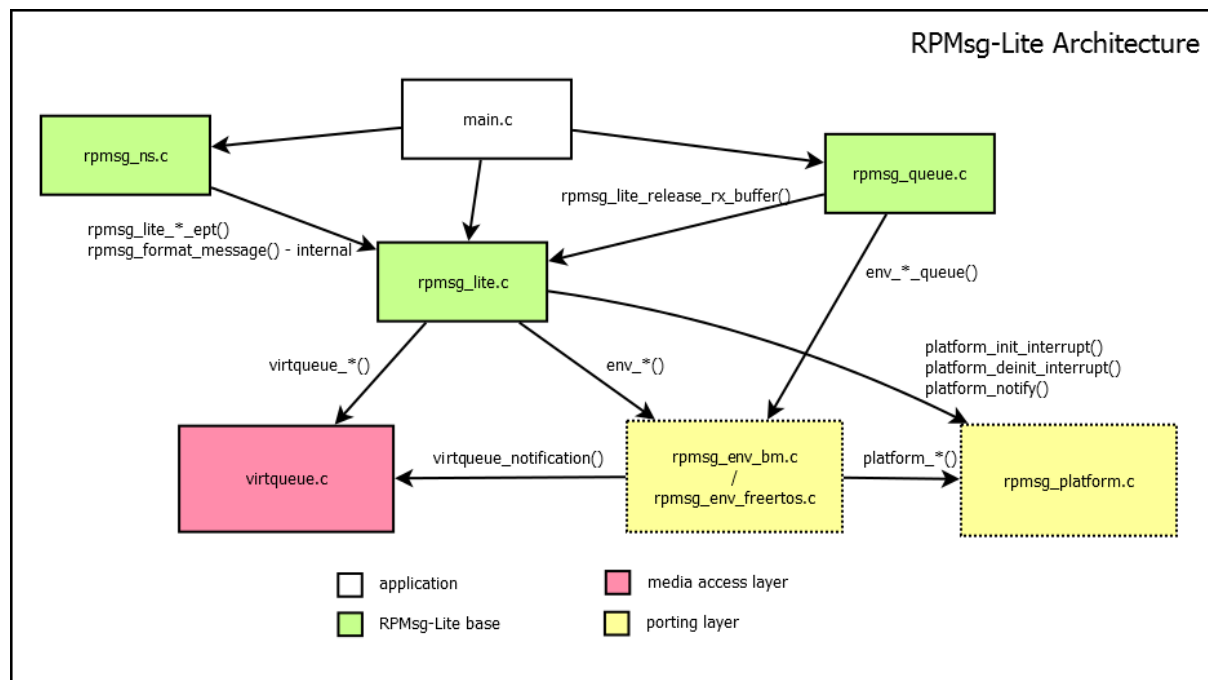
Small MCU-based systems often do not implement dynamic memory allocation. The creation of static API in RPMsg-Lite enables another reduction of resource usage. Not only does the dynamic allocation adds another 5 KB of code size, but also communication is slower and less deterministic, which is a property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

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

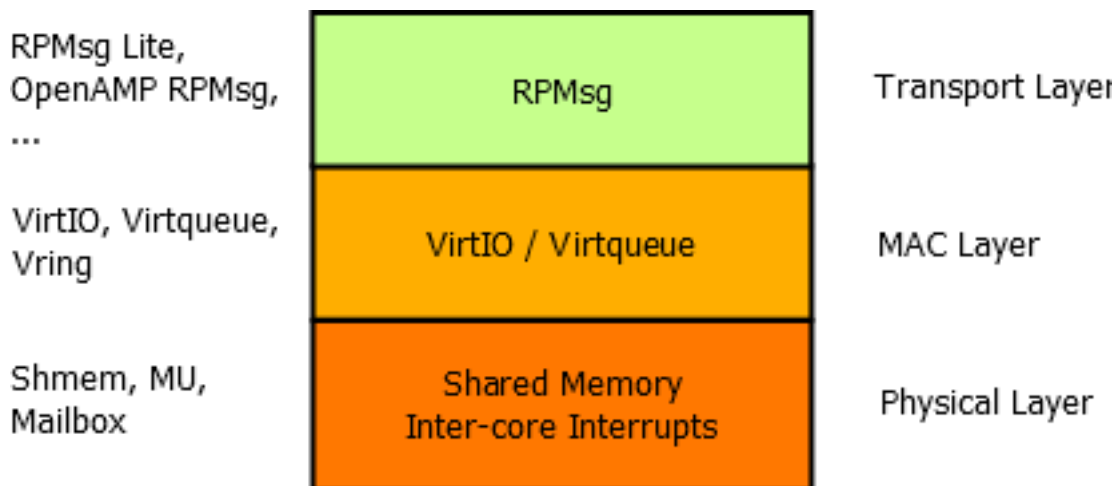
Implementation The implementation of RPMsg-Lite can be divided into three sub-components, from which two are optional. The core component is situated in `rpmsg_lite.c`. Two optional components are used to implement a blocking receive API (in `rpmsg_queue.c`) and dynamic “named” endpoint creation and deletion announcement service (in `rpmsg_ns.c`).

The actual “media access” layer is implemented in `virtqueue.c`, which is one of the few files shared with the OpenAMP implementation. This layer mainly defines the shared memory model, and internally defines used components such as `vring` or `virtqueue`.

The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. (The bare metal environment already exists and is implemented in `rpmsg_env_bm.c`, and the FreeRTOS environment is implemented in `rpmsg_env_freertos.c` etc.) Only the source file, which matches the used environment, is included in the target application project. The second sublayer is implemented in `rpmsg_platform.c` and defines low-level functions for interrupt enabling, disabling, and triggering mainly. The situation is described in the following figure:



RPMsg-Lite core sub-component This subcomponent implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer. This is realized by using so-called endpoints. Each endpoint can be assigned a different receive callback function. However, it is important to notice that the callback is executed in an interrupt environment in current design. Therefore, certain actions like memory allocation are discouraged to execute in the callback. The following figure shows the role of RPMsg in an ISO/OSI-like layered model:



Queue sub-component (optional) This subcomponent is optional and requires implementation of the `env_*_queue()` functions in the environment porting layer. It uses a blocking receive API, which is common in RTOS-environments. It supports both copy and nocopy blocking receive functions.

Name Service sub-component (optional) This subcomponent is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about “named” endpoint (in other words, channel) creation or deletion and to receive these announcement taking any user-defined action

in an application callback. The endpoint address used to receive name service announcements is arbitrarily fixed to be 53 (0x35).

Usage The application should put the `/rpsmsg_lite/lib/include` directory to the include path and in the application, include either the `rpsmsg_lite.h` header file, or optionally also include the `rpsmsg_queue.h` and/or `rpsmsg_ns.h` files. Both porting sublayers should be provided for you by NXP, but if you plan to use your own RTOS, all you need to do is to implement your own environment layer (in other words, `rpsmsg_env_myrtos.c`) and to include it in the project build.

The initialization of the stack is done by calling the `rpsmsg_lite_master_init()` on the master side and the `rpsmsg_lite_remote_init()` on the remote side. This initialization function must be called prior to any RPMsg-Lite API call. After the init, it is wise to create a communication endpoint, otherwise communication is not possible. This can be done by calling the `rpsmsg_lite_create_ept()` function. It optionally accepts a last argument, where an internal context of the endpoint is created, just in case the `RL_USE_STATIC_API` option is set to 1. If not, the stack internally calls `env_alloc()` to allocate dynamic memory for it. In case a callback-based receiving is to be used, an ISR-callback is registered to each new endpoint with user-defined callback data pointer. If a blocking receive is desired (in case of RTOS environment), the `rpsmsg_queue_create()` function must be called before calling `rpsmsg_lite_create_ept()`. The queue handle is passed to the endpoint creation function as a callback data argument and the callback function is set to `rpsmsg_queue_rx_cb()`. Then, it is possible to use `rpsmsg_queue_receive()` function to listen on a queue object for incoming messages. The `rpsmsg_lite_send()` function is used to send messages to the other side.

The RPMsg-Lite also implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

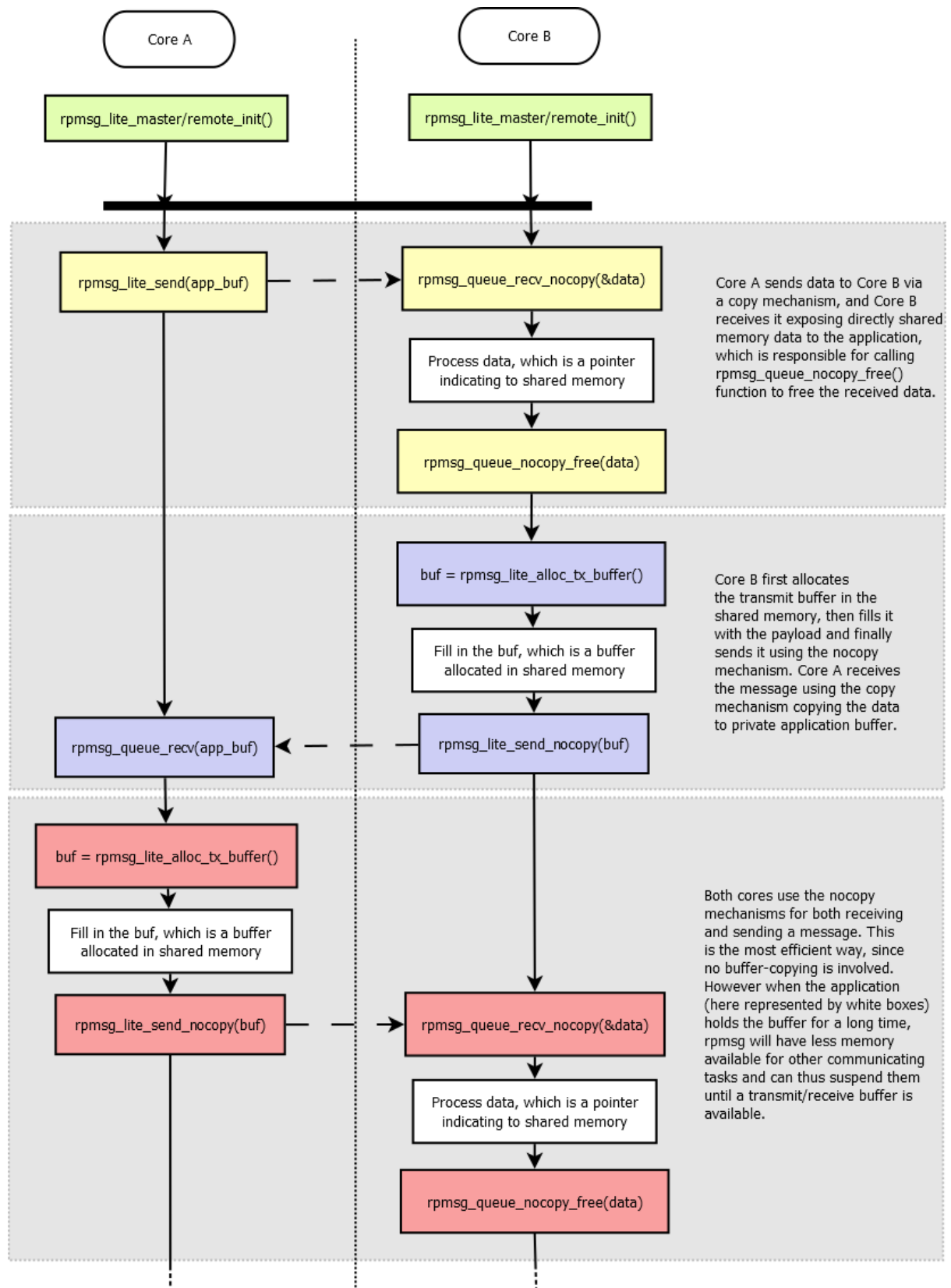
no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the `rpsmsg_lite_alloc_tx_buffer()` function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the `rpsmsg_lite_alloc_tx_buffer()` size output parameter).
- Call the `rpsmsg_lite_send_nocopy()` function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the `rpsmsg_lite_send_nocopy()` function description below.

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

- Call the `rpsmsg_queue_rcv_nocopy()` function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the `rpsmsg_queue_nocopy_free()` function to release the virtio buffer and to make it available for the next data transfer.

The user is responsible for destroying any RPMsg-Lite objects he has created in case of deinitialization. In order to do this, the function `rpsmsg_queue_destroy()` is used to destroy a queue, `rpsmsg_lite_destroy_ept()` is used to destroy an endpoint and finally, `rpsmsg_lite_deinit()` is used to deinitialize the RPMsg-Lite intercore communication stack. Deinitialize all endpoints using a queue before deinitializing the queue. Otherwise, you are actively invalidating the used queue handle, which is not allowed. RPMsg-Lite does not check this internally, since its main aim is to be lightweight.



Examples RPMsg_Lite multicore examples are part of NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages. To get the board list with multicore support (RPMsg_Lite included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded,

see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for RPMsg_Lite multicore examples with 'rpmsg_lite_' name prefix.

Another way of getting NXP MCUXpressoSDK RPMsg_Lite multicore examples is using the [mcuxsdk-manifests](#) Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk-manifests repo in [readme section](#). Once done the armgcc rpmsg_lite examples can be found in

mcuxsdk/examples/_<board_name>/multicore_examples

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the RPMsg_Lite examples use the 'rpmsg_lite_' name prefix.

Notes

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

- rpmsg_env_bm.c
- rpmsg_env_freertos.c
- rpmsg_env_xos.c
- rpmsg_env_threadx.c

The rest of environment layers has been created and used in some experimental projects, it has been running well at the time of creation but due to the lack of unit testing there is no guarantee it is still fully functional.

Shared memory configuration It is important to correctly initialize/configure the shared memory for data exchange in the application. The shared memory must be accessible from both the master and the remote core and it needs to be configured as Non-Cacheable memory. Dedicated shared memory section in linker file is also a good practise, it is recommended to use linker files from MCUXpressoSDK packages for NXP devices based applications. It needs to be ensured no other application part/component is unintentionally accessing this part of memory.

Configuration options The RPMsg-Lite can be configured at the compile time. The default configuration is defined in the rpmsg_default_config.h header file. This configuration can be customized by the user by including rpmsg_config.h file with custom settings. The following table summarizes all possible RPMsg-Lite configuration options.

Config- uration option	De- fault value	Usage
RL_MS_PE (1)		Delay in milliseconds used in non-blocking API functions for polling.
RL_BUFFE (496)		Size of the buffer payload, it must be equal to (240, 496, 1008, ...) [$2^n - 16$]
RL_BUFFE (2)		Number of the buffers, it must be power of two (2, 4, ...)
RL_API_H (1)		Zero-copy API functions enabled/disabled.
RL_USE_S' (0)		Static API functions (no dynamic allocation) enabled/disabled.
RL_USE_D (0)		Memory cache management of shared memory. Use in case of data cache is enabled for shared memory.
RL_CLEAF (0)		Clearing used buffers before returning back to the pool of free buffers enabled/disabled.
RL_USE_M (0)		When enabled IPC interrupts are managed by the Multicore Manager (IPC interrupts router), when disabled RPMsg-Lite manages IPC interrupts by itself.
RL_USE_E (0)		When enabled the environment layer uses its own context. Required for some environments (QNX). The default value is 0 (no context, saves some RAM).
RL_DEBU (0)		When enabled buffer pointers passed to <code>rpmsg_lite_send_nocopy()</code> and <code>rpmsg_lite_release_rx_buffer()</code> functions (enabled by <code>RL_API_HAS_ZEROCOPY</code> config) are checked to avoid passing invalid buffer pointer. The default value is 0 (disabled). Do not use in RPMsg-Lite to Linux configuration.
RL_ALLO (0)		When enabled the opposite side is notified each time received buffers are consumed and put into the queue of available buffers. Enable this option in RPMsg-Lite to Linux configuration to allow unblocking of the Linux blocking send. The default value is 0 (RPMsg-Lite to RPMsg-Lite communication).
RL_ALLO (0)		It allows to define custom shared memory configuration and replacing the shared memory related global settings from <code>rpmsg_config.h</code> . This is useful when multiple instances are running in parallel but different shared memory arrangement (vring size & alignment, buffers size & count) is required. The default value is 0 (all RPMsg-Lite instances use the same shared memory arrangement as defined by common config macros).
RL_ASSERT	see <code>rpmsg</code>	Assert implementation.

How to format rpmsg-lite code To format code, use the application developed by Google, named *clang-format*. This tool is part of the [llvm](#) project. Currently, the clang-format 10.0.0 version is used for rpmsg-lite. The set of style settings used for clang-format is defined in the `.clang-format` file, placed in a root of the rpmsg-lite directory where Python script `run_clang_format.py` can be executed. This script executes the application named *clang-format.exe*. You need to have the path of this application in the OS's environment path, or you need to change the script.

References

[1] M. Novak, M. Cingel, **Lockless Shared Memory Based Multicore Communication Protocol**
Copyright © 2016 Freescale Semiconductor, Inc. Copyright © 2016-2025 NXP

Changelog RPMSG-Lite All notable changes to this project will be documented in this file. The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Fixed

- Fixed CERT-C INT31-C violation in platform_notify function in rpmsg_platform.c for imxrt700_m33, imxrt700_hifi4, imxrt700_hifi1 platforms

v5.2.0

Added

- Add MCXL20 porting layer and unit testing
- New utility macro RL_CALCULATE_BUFFER_COUNT_DOWN_SAFE to safely determine maximum buffer count within shared memory while preventing integer underflow.
- RT700 platform add support for MCMGR in DSPs

Changed

- Change rpmsg_platform.c to support new MCMGR API
- Improved input validation in initialization functions to properly handle insufficient memory size conditions.
- Refactored repeated buffer count calculation pattern for better code maintainability.
- To make sure that remote has already registered IRQ there is required App level IPC mechanism to notify master about it

Fixed

- Fixed env_wait_for_link_up function to handle timeout in link state checks for baremetal and qnx environment, RL_BLOCK mode can be used to wait indefinitely.
- Fixed CERT-C INT31-C violation by adding compile-time check to ensure RL_PLATFORM_HIGHEST_LINK_ID remains within safe range for 16-bit casting in virtqueue ID creation.
- Fixed CERT-C INT30-C violations by adding protection against unsigned integer underflow in shared memory calculations, specifically in shmem_length - (uint32_t)RL_VRING_OVERHEAD and shmem_length - 2U * shmem_config.vring_size expressions.
- Fixed CERT INT31-C violation in platform_interrupt_disable() and similar functions by replacing unsafe cast from uint32_t to int32_t with a return of 0 constant.
- Fixed unsigned integer underflow in rpmsg_lite_alloc_tx_buffer() where subtracting header size from buffer size could wrap around if buffer was too small, potentially leading to incorrect buffer sizing.
- Fixed CERT-C INT31-C violation in rpmsg_lite.c where size parameter was cast from uint32_t to uint16_t without proper validation.
 - Applied consistent masking approach to both size and flags parameters: (uint16_t)(value & 0xFFFFU).
 - This fix prevents potential data loss when size values exceed 65535.

- Fixed CERT INT31-C violation in `env_memset` functions by explicitly converting `int32_t` values to unsigned char using bit masking. This prevents potential data loss or misinterpretation when passing values outside the unsigned char range (0-255) to the standard `memset()` function.
- Fixed CERT-C INT31-C violations in RPMsg-Lite environment porting: Added validation checks for signed-to-unsigned integer conversions to prevent data loss and misinterpretation.
 - `rpmsg_env_freertos.c`: Added validation before converting `int32_t` to `UBaseType_t`.
 - `rpmsg_env_qnx.c`: Fixed format string and added validation before assigning to `mqstat` fields.
 - `rpmsg_env_threadx.c`: Added validation to prevent integer overflow and negative values.
 - `rpmsg_env_xos.c`: Added range checking before casting to `uint16_t`.
 - `rpmsg_env_zephyr.c`: Added validation before passing values to `k_msgq_init`.
- Fixed a CERT INT31-C compliance issue in `env_get_current_queue_size()` function where an unsigned queue count was cast to a signed `int32_t` without proper validation, which could lead to lost or misinterpreted data if queue size exceeded `INT32_MAX`.
- Fixed CERT INT31-C violation in `rpmsg_platform.c` where `memcmp()` return value (signed int) was compared with unsigned constant without proper type handling.
- Fixed CERT INT31-C violation in `rpmsg_platform.c` where casting from `uint32_t` to `uint16_t` could potentially result in data loss. Changed length variable type from `uint16_t` to `uint32_t` to properly handle memory address differences without truncation.
- Fixed potential integer overflow in `env_sleep_msec()` function in ThreadX environment implementation by rearranging calculation order in the sleep duration formula.
- Fixed CERT-C INT31-C violation in RPMsg-Lite where bitwise NOT operations on integer constants were performed in signed integer context before being cast to unsigned. This could potentially lead to misinterpreted data on `imx943` platform.
- Added `RL_MAX_BUFFER_COUNT` (32768U) and `RL_MAX_VRING_ALIGN` (65536U) limit to ensure alignment values cannot contribute to integer overflow
- Fixed CERT INT31-C violation in `vring_need_event()`, added cast to `uint16_t` for each operand.

v5.1.4 - 27-Mar-2025

Added

- Add KW43B43 porting layer

Changed

- Doxygen bump to version 1.9.6

v5.1.3 - 13-Jan-2025

Added

- Memory cache management of shared memory. Enable with `#define RL_USE_DCACHE (1)` in `rpmsg_config.h` in case of data cache is used.
- Cmake/Kconfig support added.
- Porting layers for imx95, imxrt700, mcmxw71x, mcmxw72x, kw47b42 added.

v5.1.2 - 08-Jul-2024**Changed**

- Zephyr-related changes.
- Minor Misra corrections.

v5.1.1 - 19-Jan-2024**Added**

- Test suite provided.
- Zephyr support added.

Changed

- Minor changes in platform and env. layers, minor test code updates.

v5.1.0 - 02-Aug-2023**Added**

- RPMsg-Lite: Added aarch64 support.

Changed

- RPMsg-Lite: Increased the queue size to $(2 * RL\_BUFFER\_COUNT)$ to cover zero copy cases.
- Code formatting using LLVM16.

Fixed

- Resolved issues in ThreadX env. layer implementation.

v5.0.0 - 19-Jan-2023**Added**

- Timeout parameter added to `rpmsg_lite_wait_for_link_up` API function.

Changed

- Improved debug check buffers implementation - instead of checking the pointer fits into shared memory check the presence in the VirtIO ring descriptors list.
- VRING_SIZE is set based on number of used buffers now (as calculated in vring_init) - updated for all platforms that are not communicating to Linux rpmsg counterpart.

Fixed

- Fixed wrong RL_VRING_OVERHEAD macro comment in platform.h files
- Misra corrections.

v4.0.0 - 20-Jun-2022

Added

- Added support for custom shared memory arrangement per the RPMsg_Lite instance.
- Introduced new rpmsg_lite_wait_for_link_up() API function - this allows to avoid using busy loops in rtos environments, GitHub PR [#21](#).

Changed

- Adjusted rpmsg_lite_is_link_up() to return RL_TRUE/RL_FALSE.

v3.2.0 - 17-Jan-2022

Added

- Added support for i.MX8 MP multicore platform.

Changed

- Improved static allocations - allow OS-specific objects being allocated statically, GitHub PR [#14](#).
- Aligned rpmsg_env_xos.c and some platform layers to latest static allocation support.

Fixed

- Minor Misra and typo corrections, GitHub PR [#19](#), [#20](#).

v3.1.2 - 16-Jul-2021

Added

- Addressed MISRA 21.6 rule violation in rpmsg_env.h (use SDK's PRINTF in MCUXpressoSDK examples, otherwise stdio printf is used).
- Added environment layers for XOS.
- Added support for i.MX RT500, i.MX RT1160 and i.MX RT1170 multicore platforms.

Fixed

- Fixed incorrect description of the `rpmsg_lite_get_endpoint_from_addr` function.

Changed

- Updated `RL_BUFFER_COUNT` documentation (issue [#10](#)).
- Updated `imxrt600_hifi4` platform layer.

v3.1.1 - 15-Jan-2021

Added

- Introduced `RL_ALLOW_CONSUMED_BUFFERS_NOTIFICATION` config option to allow opposite side notification sending each time received buffers are consumed and put into the queue of available buffers.
- Added environment layers for Threadx.
- Added support for i.MX8QM multicore platform.

Changed

- Several MISRA C-2012 violations addressed.

v3.1.0 - 22-Jul-2020

Added

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed (7.4).
- Fixed missing lock in `rpmsg_lite_rx_callback()` for QNX env.
- Correction of `rpmsg_lite_instance` structure members description.
- Address -Waddress-of-packed-member warnings in GCC9.

Changed

- Clang update to v10.0.0, code re-formatted.

v3.0.0 - 20-Dec-2019

Added

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed, incl. data types consolidation.
- Code formatted.

v2.2.0 - 20-Mar-2019

Added

- Added configuration macro `RL_DEBUG_CHECK_BUFFERS`.
- Several MISRA violations fixed.
- Added environment layers for QNX and Zephyr.
- Allow environment context required for some environment (controlled by the `RL_USE_ENVIRONMENT_CONTEXT` configuration macro).
- Data types consolidation.

v1.1.0 - 28-Apr-2017

Added

- Supporting i.MX6SX and i.MX7D MPU platforms.
- Supporting LPC5411x MCU platform.
- Baremetal and FreeRTOS support.
- Support of copy and zero-copy transfer.
- Support of static API (without dynamic allocations).

Multicore Manager

MCUXpresso SDK : `mcuxsdk-middleware-mcmgr` (Multicore Manager)

Overview This repository is for MCUXpresso SDK Multicore Manager middleware delivery and it contains Multicore Manager component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run Multicore Manager examples that are based on `mcux-sdk-middleware-mcmgr` component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

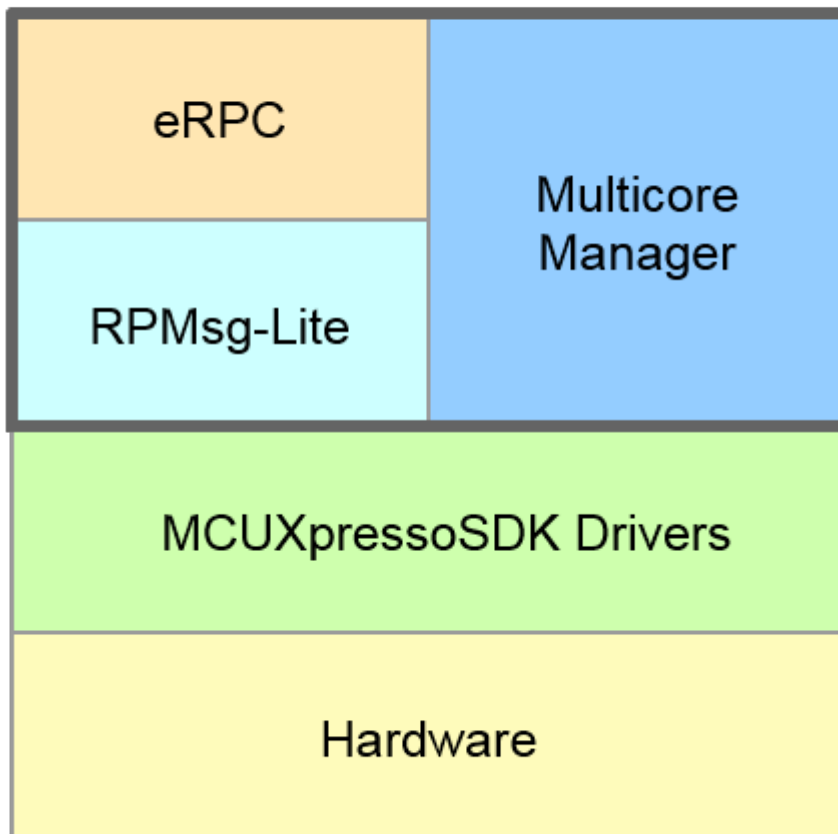
Visit [Multicore Manager - Documentation](#) to review details on the contents in this sub-repo.

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the mcmgr project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems. This library is distributed as a part of the Multicore SDK (MCSDK). Together, the MCSDK and the MCUXpresso SDK (SDK) form a framework for development of software for NXP multicore devices.

The MCMGR component is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr directory.



The Multicore Manager provides the following major functions:

- Maintains information about all cores in system.
- Secondary/auxiliary core(s) startup and shutdown.
- Remote core monitoring and event handling.

For Further API documentation, please look at doxygen documentation at: <https://nxp-mcuxpresso.github.io/mcux-mcmgr/>

Usage of the MCMGR software component The main use case of MCMGR is the secondary/auxiliary core start. This functionality is performed by the public API function.

Example of MCMGR usage to start secondary core:

```
#include "mcmgr.h"

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

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass "1" as startup data,
    ↪starting synchronously. */
    MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 1, kMCMGR_Start_Synchronous);
    .
    .
    .
    /* Stop secondary core execution. */
    MCMGR_StopCore(kMCMGR_Core1);
}
```

Some platforms allow stopping and re-starting the secondary core application again, using the MCMGR_StopCore / MCMGR_StartCore API calls. It is necessary to ensure the initially loaded image is not corrupted before re-starting, especially if it deals with the RAM target. Cache coherence has to be considered/ensured as well.

Another important MCMGR feature is the ability for remote core monitoring and handling of events such as reset, exception, and application events. Application-specific callback functions for events are registered by the MCMGR_RegisterEvent() API. Triggering these events is done using the MCMGR_TriggerEvent() API. mcmgr_event_type_t enums all possible event types.

An example of MCMGR usage for remote core monitoring and event handling. Code for the primary side:

```
#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA (1)
#define APP_NUMBER_OF_CORES (2)
#define APP_SECONDARY_CORE kMCMGR_Core1

/* Callback function registered via the MCMGR_RegisterEvent() and triggered by MCMGR_TriggerEvent()
↪called on the secondary core side */
void RPSGRemoteReadyEventHandler(mcmgr_core_t coreNum, uint16_t eventData, void *context)
{
    uint16_t *data = &((uint16_t *)context)[coreNum];

    *data = eventData;
}

void main()
{
    uint16_t RPSGRemoteReadyEventData[NUMBER_OF_CORES] = {0};

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

(continues on next page)

(continued from previous page)

```

/* Initialize MCMGR, install generic event handlers */
MCMGR_Init();

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

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

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

```

Code for the secondary side:

```

#include "mcmgr.h"

#define APP_RPSMSG_READY_EVENT_DATA (1)

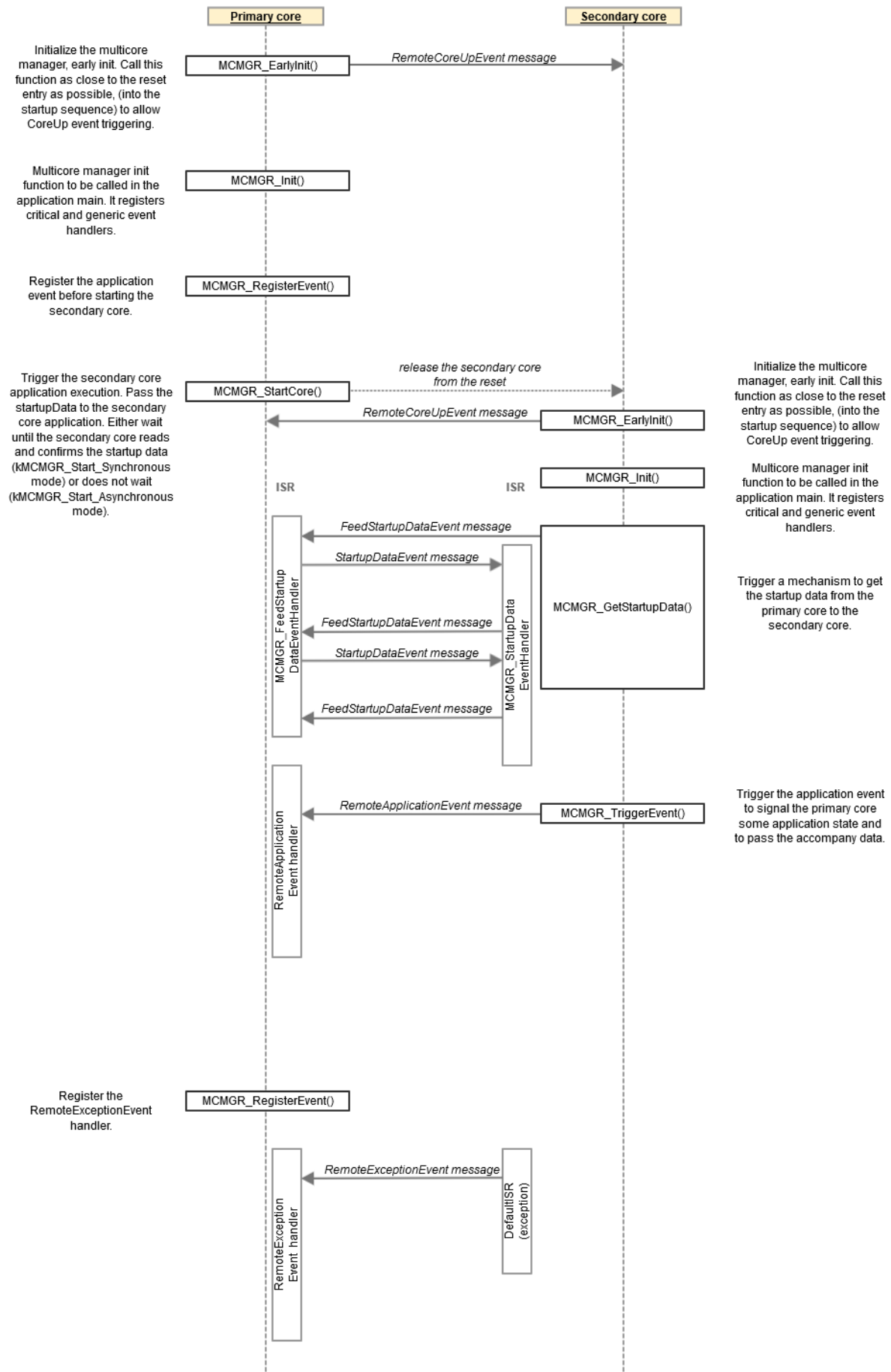
void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
    .
    .
    .

    /* Signal the to other core that we are ready by triggering the event and passing the APP_RPSMSG_
    ↳READY_EVENT_DATA */
    MCMGR_TriggerEvent(kMCMGR_Core0, kMCMGR_RemoteApplicationEvent, APP_RPSMSG_
    ↳READY_EVENT_DATA);
    .
    .
    .
}

```

MCMGR Data Exchange Diagram The following picture shows how the handshakes are supposed to work between the two cores in the MCMGR software.



Changelog Multicore Manager All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Added

Fixed

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

v5.0.0

Added

- Added MCMGR_BUSY_POLL_COUNT macro to prevent infinite polling loops in MCMGR operations.
- Implemented timeout mechanism for all polling loops in MCMGR code.
- Added support to handle more then two cores. Breaking API change by adding parameter `coreNum` specifying core number in functions bellow.
 - MCMGR_GetStartupData(uint32_t *startupData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEvent(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEventForce(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - typedef void (*mcmgr_event_callback_t)(uint16_t data, void *context, mcmgr_core_t coreNum);

When registering the event with function MCMGR_RegisterEvent() user now needs to provide `callbackData` pointer to array of elements per every core in system (see README.md for example). In case of systems with only two cores the `coreNum` in callback can be ignored as events can arrive only from one core. Please see Porting guide for more details: Porting-GuideTo_v5.md

- Updated all porting files to support new MCMGR API.
- Added new platform specific include file `mcmgr_platform.h`. It will contain common platform specific macros that can be then used in `mcmgr` and application. e.g. platform core count MCMGR_CORECOUNT 4.
- Move all header files to new `inc` directory.
- Added new platform-specific include files `inc/platform/<platform_name>/mcmgr_platform.h`.

Added

- Add MCXL20 porting layer and unit testing

v4.1.7

Fixed

- `mcmgr_stop_core_internal()` function now returns `kStatus_MCMGR_NotImplemented` status code instead of `kStatus_MCMGR_Success` when device does not support stop of secondary core. Ports affected: kw32w1, kw45b41, kw45b42, mcxw716, mcxw727.

[v4.1.6]

Added

- Multicore Manager moved to standalone repository.
- Add porting layers for imxrt700, mcmxw727, kw47b42.
- New `MCMGR_ProcessDeferredRxIsr()` API added.

[v4.1.5]

Added

- Add notification into `MCMGR_EarlyInit` and `mcmgr_early_init_internal` functions to avoid using uninitialized data in their implementations.

[v4.1.4]

Fixed

- Avoid calling tx isr callbacks when respective Messaging Unit Transmit Interrupt Enable flag is not set in the CR/TCR register.
- Messaging Unit RX and status registers are cleared after the initialization.

[v4.1.3]

Added

- Add porting layers for imxrt1180.

Fixed

- `mu_isr()` updated to avoid calling tx isr callbacks when respective Transmit Interrupt Enable flag is not set in the CR/TCR register.
- `mcmgr_mu_internal.c` code adaptation to new supported SoCs.

[v4.1.2]

Fixed

- Update `mcmgr_stop_core_internal()` implementations to set core state to `kMCMGR_ResetCoreState`.

[v4.1.0]

Fixed

- Code adjustments to address MISRA C-2012 Rules

[v4.0.3]

Fixed

- Documentation updated to describe handshaking in a graphic form.
- Minor code adjustments based on static analysis tool findings

[v4.0.2]

Fixed

- Align porting layers to the updated MCUXpressoSDK feature files.

[v4.0.1]

Fixed

- Code formatting, removed unused code

[v4.0.0]

Added

- Add new MCMGR_TriggerEventForce() API.

[v3.0.0]

Removed

- Removed MCMGR_LoadApp(), MCMGR_MapAddress() and MCMGR_SignalReady()

Modified

- Modified MCMGR_GetStartupData()

Added

- Added MCMGR_EarlyInit(), MCMGR_RegisterEvent() and MCMGR_TriggerEvent()
- Added the ability for remote core monitoring and event handling

[v2.0.1]

Fixed

- Updated to be Misra compliant.

[v2.0.0]

Added

- Support for lpcxpresso54114 board.

[v1.1.0]

Fixed

- Ported to KSDK 2.0.0.

[v1.0.0]

Added

- Initial release.

eRPC

MCUXpresso SDK : mcuxsdk-middleware-erpc

Overview This repository is for MCUXpresso SDK eRPC middleware delivery and it contains eRPC component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository [mcuxsdk](#) for the complete delivery of MCUXpresso SDK to be able to build and run eRPC examples that are based on mcux-sdk-middleware-erpc component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

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

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest [Getting Started with SDK - Detailed Installation Instructions](#)

Contribution We welcome and encourage the community to submit patches directly to the eRPC project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

eRPC

- [MCUXpresso SDK : mcuxsdk-middleware-erpc](#)

- [Overview](#)
- [Documentation](#)
- [Setup](#)
- [Contribution](#)

- [eRPC](#)

- [About](#)
- [Releases](#)
 - * [Edge releases](#)
- [Documentation](#)
- [Examples](#)
- [References](#)
- [Directories](#)
- [Building and installing](#)
 - * [Requirements](#)
 - [Windows](#)
 - [Mac OS X](#)
 - * [Building](#)
 - [CMake and KConfig](#)
 - [Make](#)
 - * [Installing for Python](#)
- [Known issues and limitations](#)
- [Code providing](#)

About

eRPC (Embedded RPC) is an open source Remote Procedure Call (RPC) system for multichip embedded systems and heterogeneous multicore SoCs.

Unlike other modern RPC systems, such as the excellent [Apache Thrift](#), eRPC distinguishes itself by being designed for tightly coupled systems, using plain C for remote functions, and having a small code size (<5kB). It is not intended for high performance distributed systems over a network.

eRPC does not force upon you any particular API style. It allows you to export existing C functions, without having to change their prototypes. (There are limits, of course.) And although the

internal infrastructure is written in C++, most users will be able to use only the simple C setup APIs shown in the examples below.

A code generator tool called `erpcgen` is included. It accepts input IDL files, having an `.erpc` extension, that have definitions of your data types and remote interfaces, and generates the shim code that handles serialization and invocation. `erpcgen` can generate either C/C++ or Python code.

Example `.erpc` file:

```
// Define a data type.
enum LEDName { kRed, kGreen, kBlue }

// An interface is a logical grouping of functions.
interface IO {
    // Simple function declaration with an empty reply.
    set_led(LEDName whichLed, bool onOrOff) -> void
}
```

Client side usage:

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* init eRPC client IO service */
    initIO_client(client_manager);

    // Now we can call the remote function to turn on the green LED.
    set_led(kGreen, true);

    /* deinit objects */
    deinitIO_client();
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* scope for client service */
    {
        /* init eRPC client IO service */
        IO_client client(client_manager);

        // Now we can call the remote function to turn on the green LED.
        client.set_led(kGreen, true);
    }

    /* deinit objects */
```

(continues on next page)

(continued from previous page)

```

    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

Server side usage:

```

// Implement the remote function.
void set_led(LEDName whichLed, bool onOrOff) {
    // implementation goes here
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    erpc_service_t service = create_IO_service();

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, service);

    // Run the server.
    erpc_server_run();

    /* deinit objects */
    destroy_IO_service(service);
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

```

// Implement the remote function.
class IO : public IO_interface
{
    /* eRPC call definition */
    void set_led(LEDName whichLed, bool onOrOff) override {
        // implementation goes here
    }
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    IO IOImpl;
    IO_service io(&IOImpl);

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, &io);

    /* poll for requests */
}

```

(continues on next page)

(continued from previous page)

```
erpc_status_t err = server.run();

/* deinit objects */
erpc_server_deinit(server);
erpc_mbf_dynamic_deinit(message_buffer_factory);
erpc_transport_tcp_deinit(transport);
}
```

A number of transports are supported, and new transport classes are easy to write.

Supported transports can be found in *erpc/erpc_c/transport* folder. E.g:

- CMSIS UART
- NXP Kinetis SPI and DSPI
- POSIX and Windows serial port
- TCP/IP (mostly for testing)
- [NXP RPMsg-Lite / RPMsg TTY](#)
- SPIdev Linux
- USB CDC
- NXP Messaging Unit

eRPC is available with an unrestrictive BSD 3-clause license. See the [LICENSE file](#) for the full license text.

Releases [eRPC releases](#)

Edge releases Edge releases can be found on [eRPC CircleCI](#) webpage. Choose build of interest, then platform target and choose ARTIFACTS tab. Here you can find binary application from chosen build.

Documentation [Documentation](#) is in the wiki section.

[eRPC Infrastructure documentation](#)

Examples *Example IDL* is available in the *examples/* folder.

Plenty of eRPC multicore and multiprocessor examples can be also found in NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages.

To get the board list with multicore support (eRPC included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded, see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for eRPC multicore examples (RPMsg_Lite or Messaging Unit transports used) or

<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples for eRPC multiprocessor examples (UART or SPI transports used).

eRPC examples use the 'erpc_' name prefix.

Another way of getting NXP MCUXpressoSDK eRPC multicore and multiprocessor examples is using the [mcux-sdk](#) Github repo. Follow the description how to use the West tool to clone and

update the mcuxsdk repo in [readme Overview section](#). Once done the armgcc eRPC examples can be found in

mcuxsdk/examples/<board_name>/multicore_examples or in

mcuxsdk/examples/<board_name>/multiprocessor_examples folders.

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the eRPC examples use the 'erpc_' name prefix.

References This section provides links to interesting erpc-based projects, articles, blogs or guides:

- [erpc \(EmbeddedRPC\) getting started notes](#)
- [ERPC Linux Local Environment Construction and Use](#)
- [The New Wio Terminal eRPC Firmware](#)

Directories *doc* - Documentation.

doxygen - Configuration and support files for running Doxygen over the eRPC C++ infrastructure and erpcgen code.

erpc_c - Holds C/C++ infrastructure for eRPC. This is the code you will include in your application.

erpc_python - Holds Python version of the eRPC infrastructure.

erpcgen - Holds source code for erpcgen and makefiles or project files to build erpcgen on Windows, Linux, and OS X.

erpcsniffer - Holds source code for erpcsniffer application.

examples - Several example IDL files.

mk - Contains common makefiles for building eRPC components.

test - Client/server tests. These tests verify the entire communications path from client to server and back.

utilities - Holds utilities which bring additional benefit to eRPC apps developers.

Building and installing These build instructions apply to host PCs and embedded Linux. For bare metal or RTOS embedded environments, you should copy the *erpc_c* directory into your application sources.

CMake and KConfig build:

It builds a static library of the eRPC C/C++ infrastructure, the *erpcgen* executable, and optionally the unit tests and examples.

CMake is compatible with gcc and clang. On Windows local MingGW downloaded by *script* can be used.

Make build:

It builds a static library of the eRPC C/C++ infrastructure, the *erpcgen* executable, and optionally the unit tests.

The makefiles are compatible with gcc or clang on Linux, OS X, and Cygwin. A Windows build of *erpcgen* using Visual Studio is also available in the *erpcgen/VisualStudio_v14* directory. There is also an Xcode project file in the *erpcgen* directory, which can be used to build *erpcgen* for OS X.

Requirements erPC now support building **erpcgen**, **erpc_lib**, **tests** and **C examples** using CMake.

Requirements when using CMake:

- **CMake** (minimal version 3.20.0)
- Generator - **Make, Ninja, ...**
- **C/C++ compiler** - **GCC, CLANG, ...**
- **Bison** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Requirements when using Make:

- **Make**
- **C/C++ compiler - GCC, CLANG, ...**
- **Binson** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Windows Related steps to build **erpcgen** using **Visual Studio** are described in `erpcgen/VisualStudio_v14/readme_erpcgen.txt`.

To install MinGW, Bison, Flex locally on Windows:

```
./install_dependencies.ps1
* \ \ \

#### Linux

\ \ \ bash
./install_dependencies.sh
```

Mandatory for case, when build for different architecture is needed

- gcc-multilib, g++-multilib

Mac OS X

```
./install_dependencies.sh
```

Building

CMake and KConfig eRPC use CMake and KConfig to configurate and build eRPC related targets. KConfig can be edited by *prj.conf* or *menuconfig* when building.

Generate project, config and build. In *erpc/* execute:

```
cmake -B ./build # in erpc/build generate cmake project
cmake --build ./build --target menuconfig # Build menuconfig and configure erpcgen, erpc_lib, tests and
→ examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

****CMake will use the system's default compilers and generator**

If you want to use Windows and locally installed MinGW, use *CMake preset* :

```
cmake --preset mingw64 # Generate project in ./build using mingw64's make and compilers
cmake --build ./build --target menuconfig # Build menuconfig and configure erpcgen, erpc_lib, tests and ↵
↵examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

Make To build the library and erpcgen, run from the repo root directory:

```
make
```

To install the library, erpcgen, and include files, run:

```
make install
```

You may need to sudo the make install.

By default this will install into /usr/local. If you want to install elsewhere, set the PREFIX environment variable. Example for installing into /opt:

```
make install PREFIX=/opt
```

List of top level Makefile targets:

- erpc: build the liberpc.a static library
- erpcgen: build the erpcgen tool
- erpcsniffer: build the sniffer tool
- test: build the unit tests under the *test* directory
- all: build all of the above
- install: install liberpc.a, erpcgen, and include files

eRPC code is validated with respect to the C++ 11 standard.

Installing for Python To install the Python infrastructure for eRPC see instructions in the *erpc python readme*.

Known issues and limitations

- Static allocations controlled by the ERPC_ALLOCATION_POLICY config macro are not fully supported yet, i.e. not all erpc objects can be allocated statically now. It deals with the ongoing process and the full static allocations support will be added in the future.

Code providing Repository on Github contains two main branches: **main** and **develop**. Code is developed on **develop** branch. Release version is created via merging **develop** branch into **main** branch.

Copyright 2014-2016 Freescale Semiconductor, Inc.

Copyright 2016-2025 NXP

eRPC Getting Started

Overview This *Getting Started User Guide* shows software developers how to use Remote Procedure Calls (RPC) in embedded multicore microcontrollers (eRPC).

The eRPC documentation is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

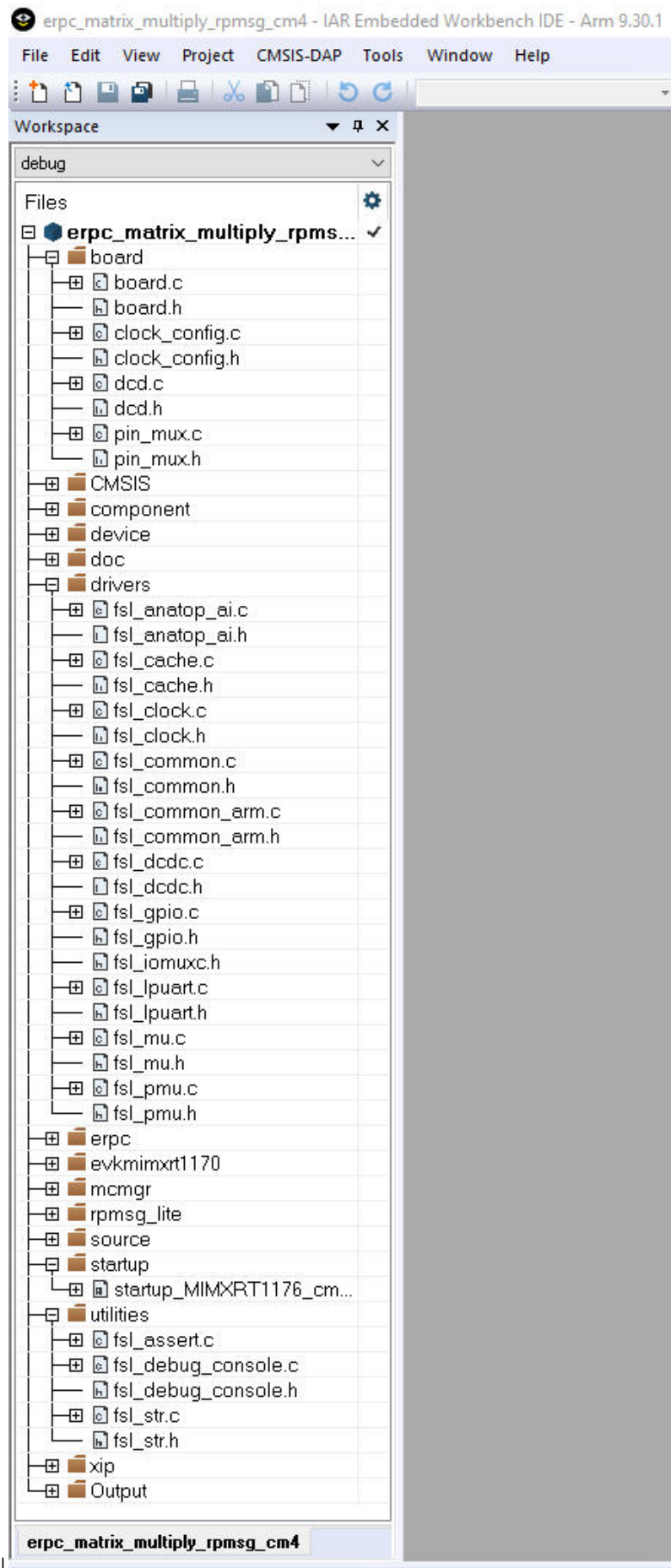
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

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

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



|

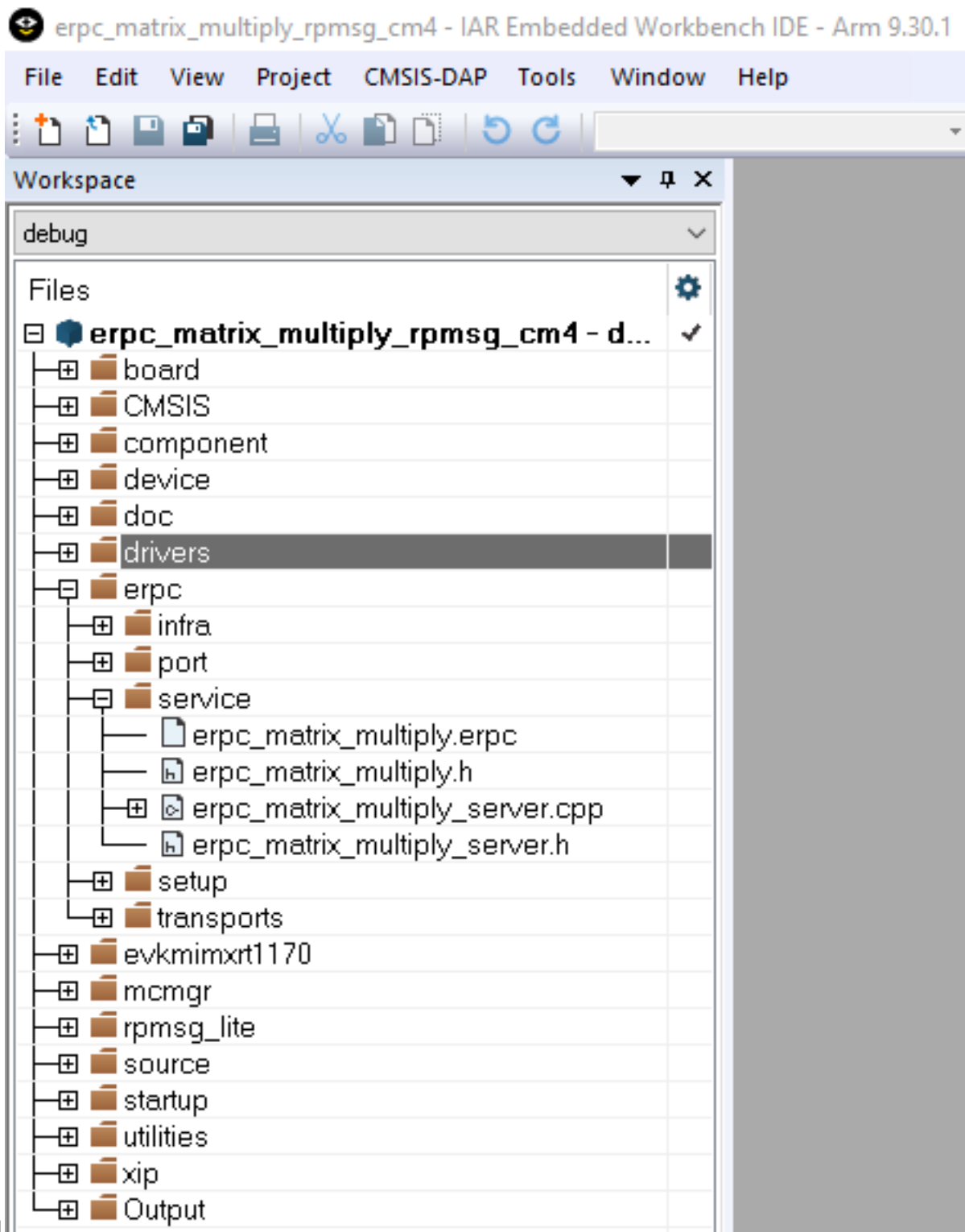
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/



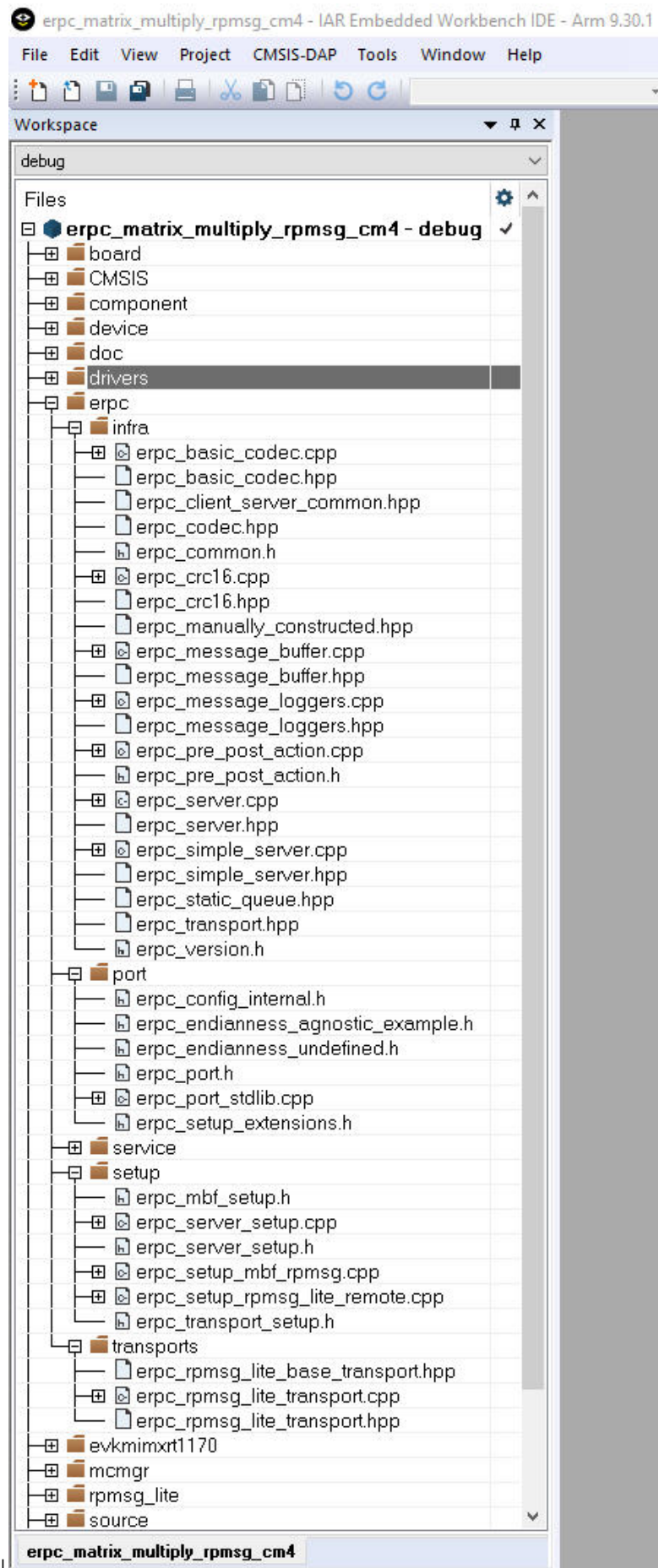
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

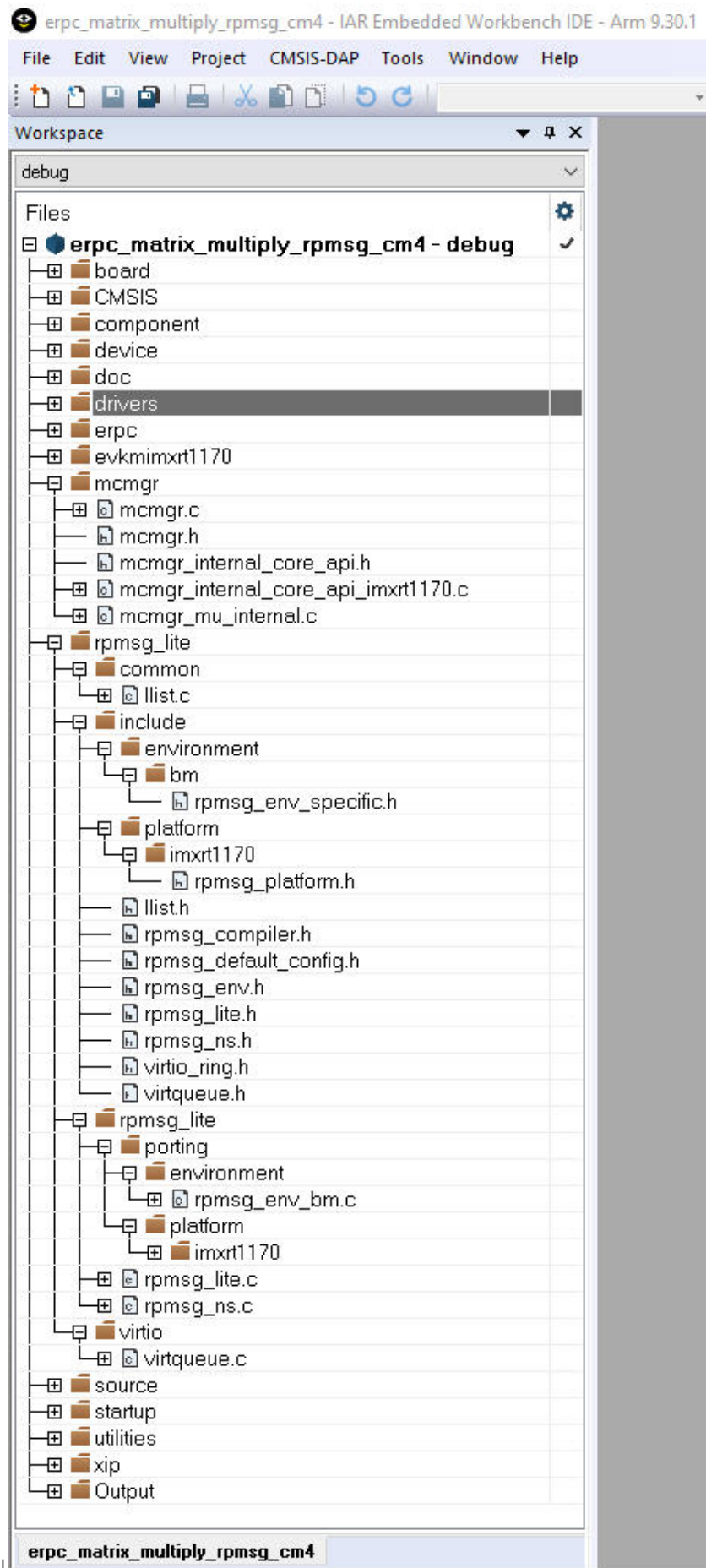
Parent topic: Multicore server application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

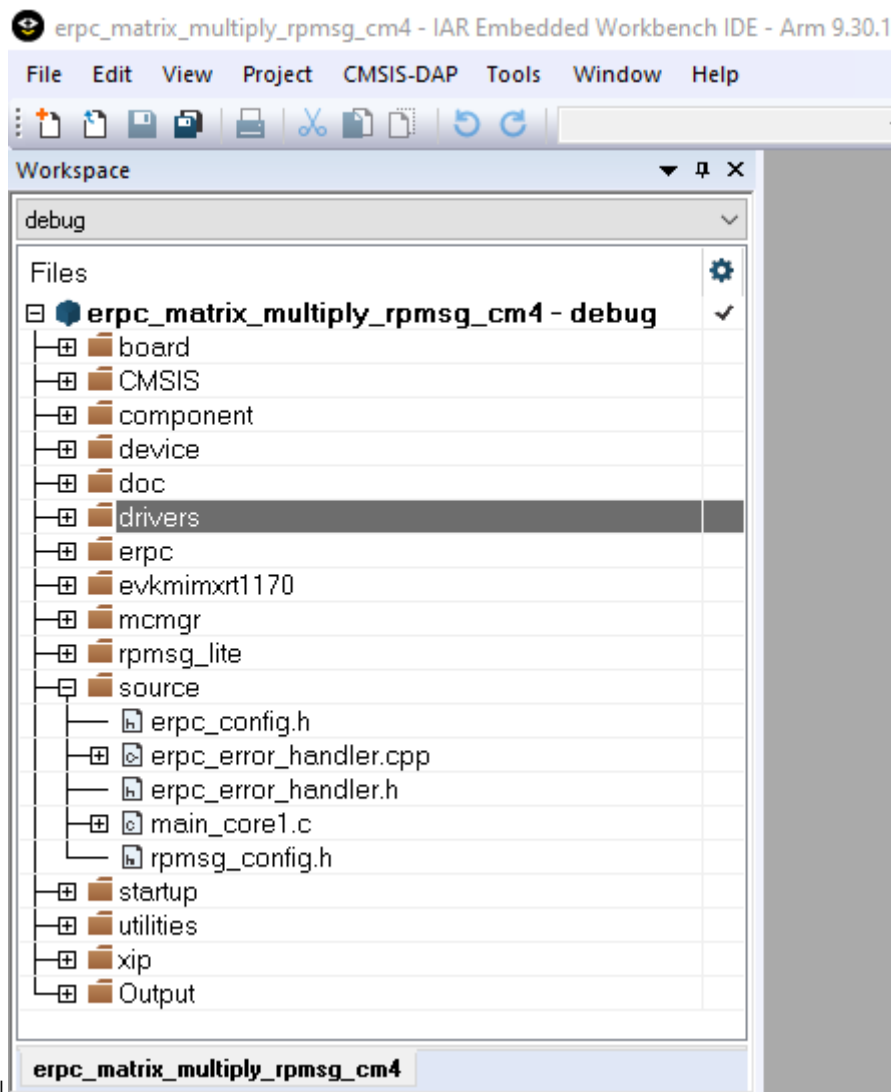
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg` folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

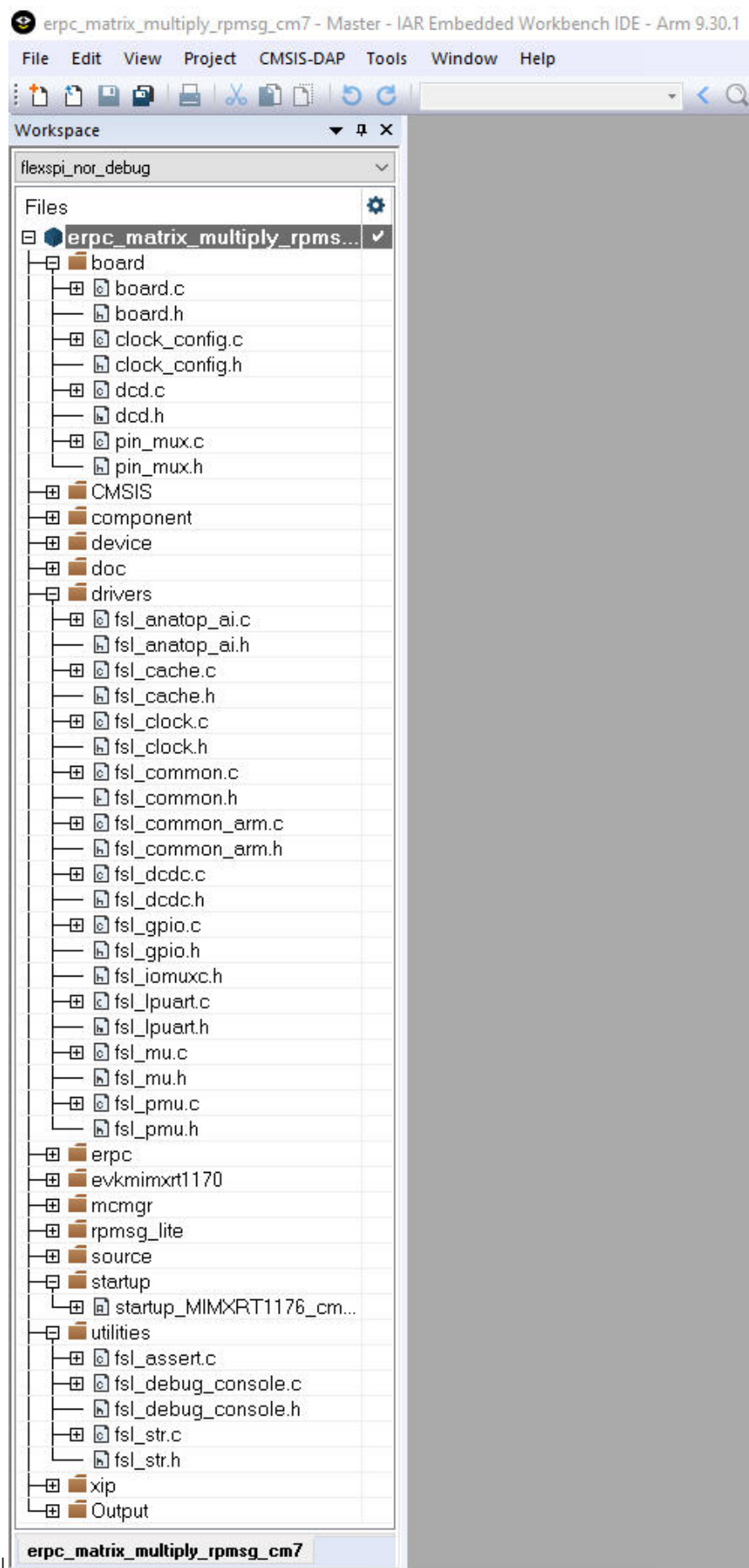
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar`

Project files for the eRPC client have the `_cm7` suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



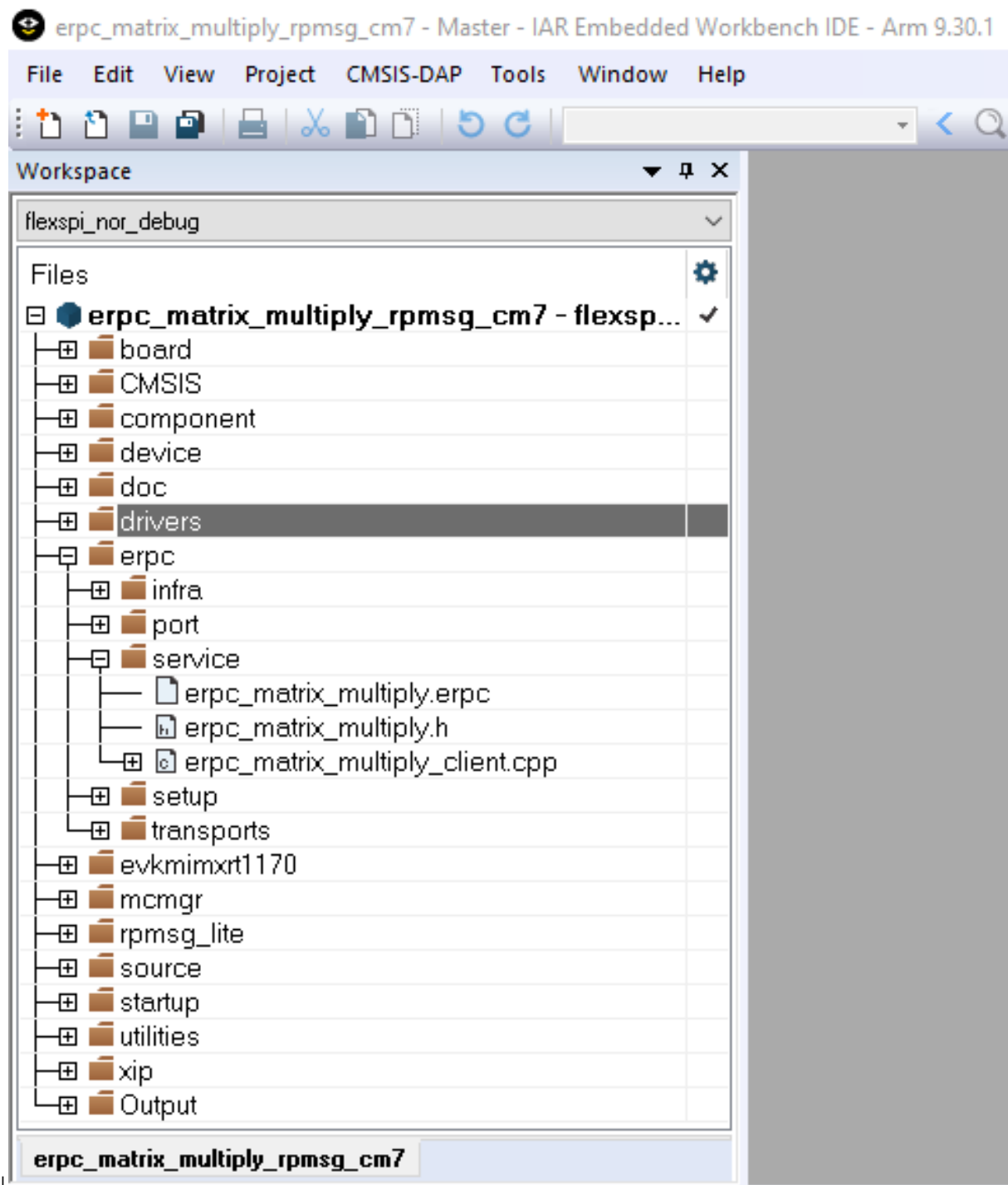
|

Parent topic:Multicore client application

Client-related generated files The client-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/ folder.



Parent topic: Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

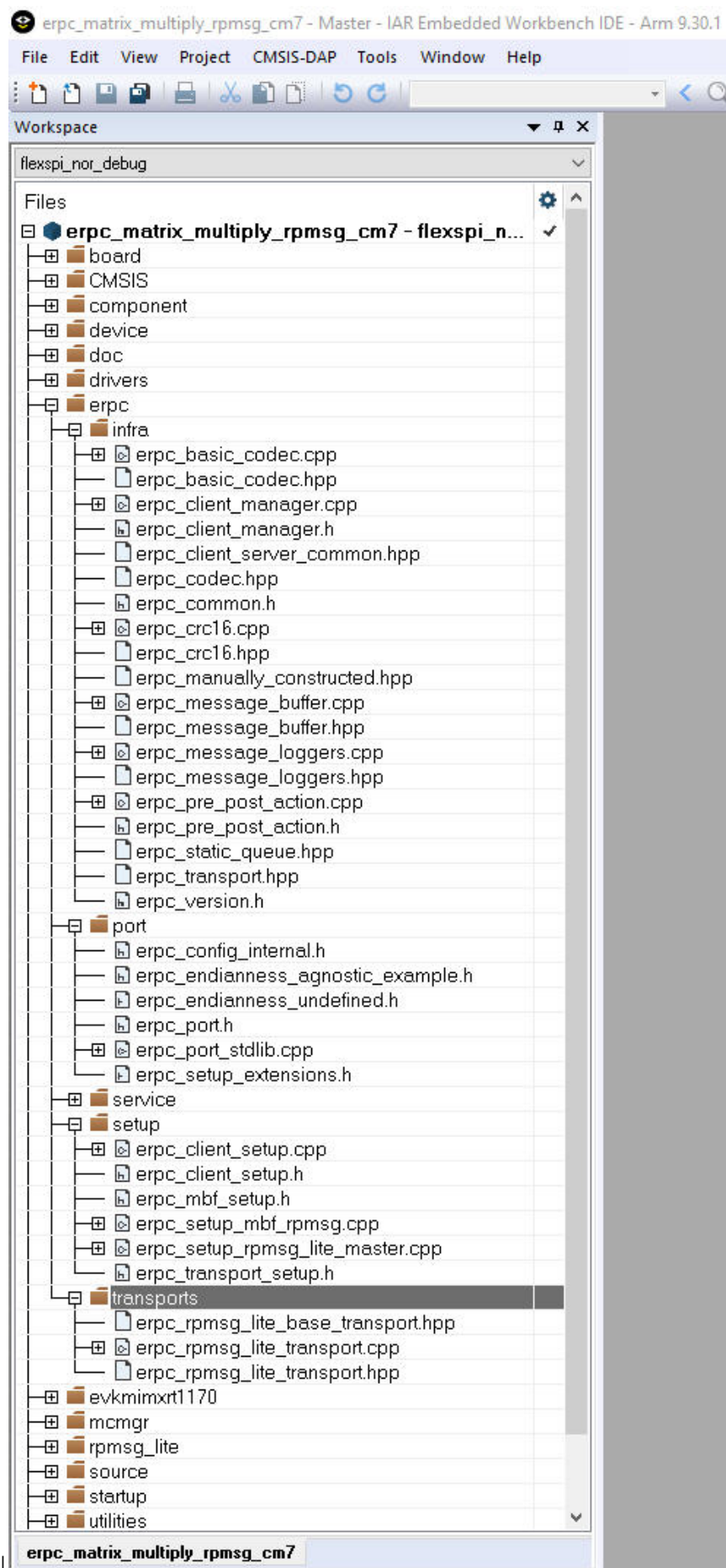
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

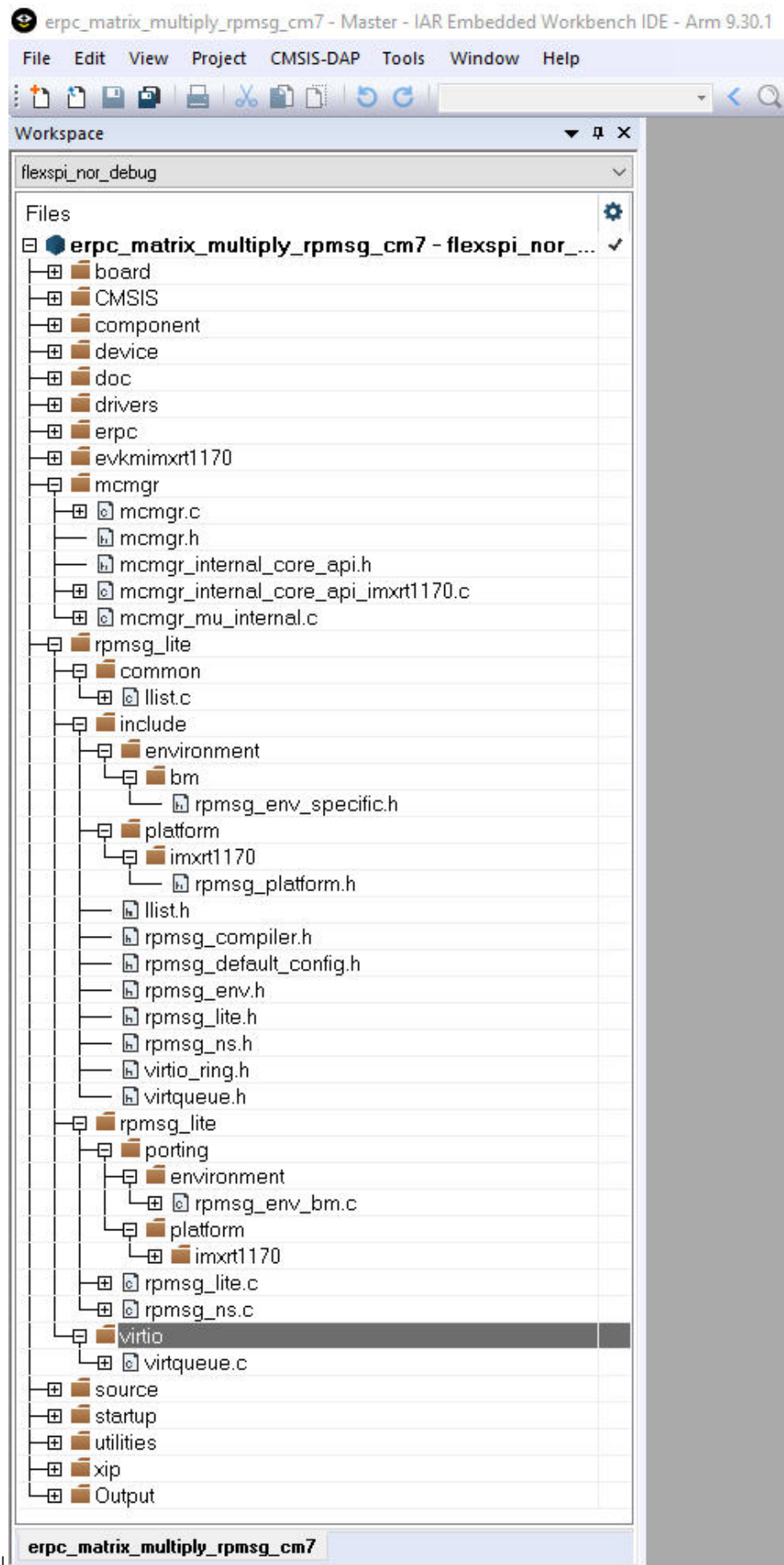
Parent topic: Multicore client application

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

<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/



Parent topic:Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

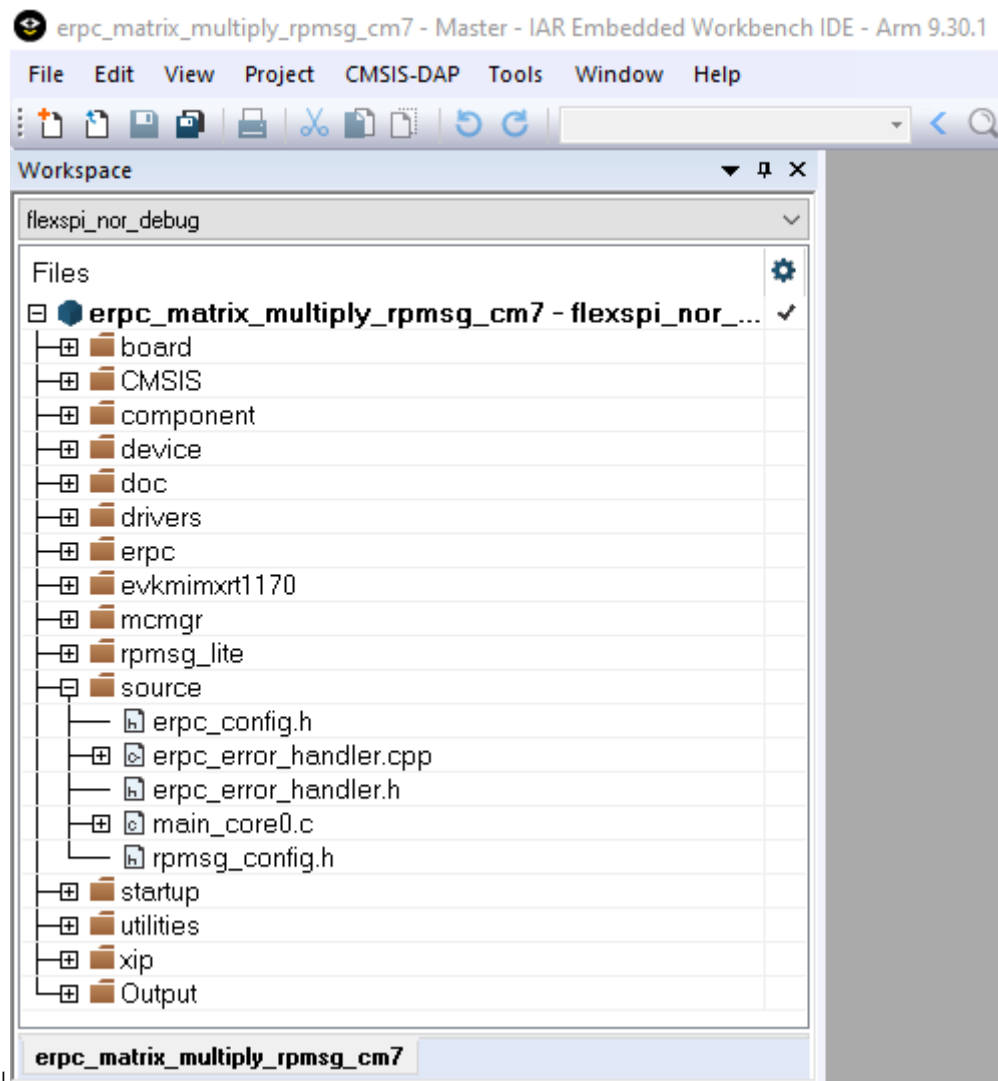
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic:Multicore client application

Parent topic:[Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPSMsg-Lite). The RPSMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.hpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.hpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

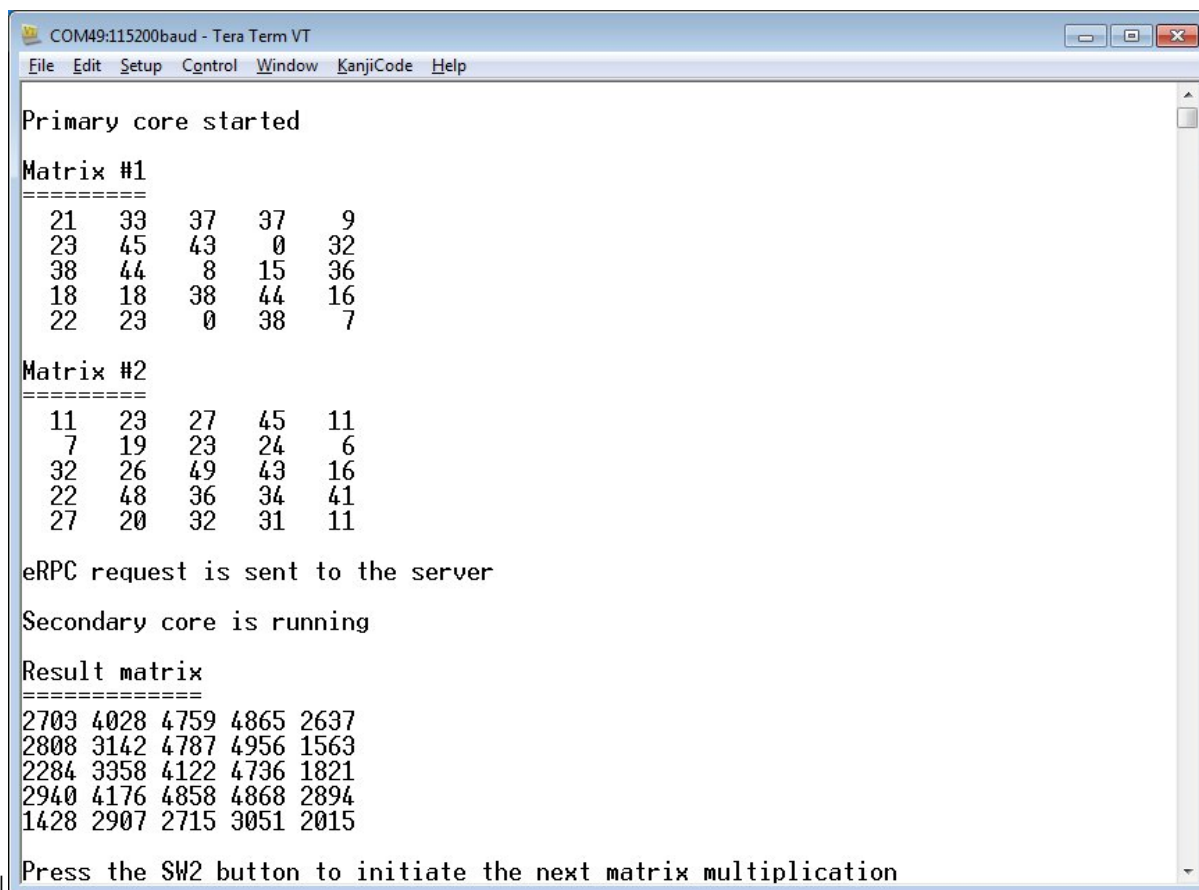
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

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

Primary core started

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

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

eRPC request is sent to the server

Secondary core is running

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

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

eRPC example This section shows how to create an example eRPC application called “Matrix multiply”, which implements one eRPC function (matrix multiply) with two function parameters (two matrices). The client-side application calls this eRPC function, and the server side performs the multiplication of received matrices. The server side then returns the result.

For example, use the NXP MIMXRT1170-EVK board as the target dual-core platform, and the IAR Embedded Workbench for ARM (EWARM) as the target IDE for developing the eRPC example.

- The primary core (CM7) runs the eRPC client.
- The secondary core (CM4) runs the eRPC server.
- RPMsg-Lite (Remote Processor Messaging Lite) is used as the eRPC transport layer.

The “Matrix multiply” application can be also run in the multi-processor setup. In other words, the eRPC client running on one SoC communicates with the eRPC server that runs on another SoC, utilizing different transport channels. It is possible to run the board-to-PC example (PC as the eRPC server and a board as the eRPC client, and vice versa) and also the board-to-board example. These multiprocessor examples are prepared for selected boards only.

| Multicore application source and project files | `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore/`
| Multiprocessor application source and project files | `<MCUXpressoSDK_install_dir>/boards/<board_name>/multi`
`<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<tr`
| | eRPC source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/` | | RPLite
source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/rplite/`

Designing the eRPC application The matrix multiply application is based on calling single eRPC function that takes 2 two-dimensional arrays as input and returns matrix multiplication results as another 2 two-dimensional array. The IDL file syntax supports arrays with the dimension length set by the number only (in the current eRPC implementation). Because of this, a variable is declared in the IDL dedicated to store information about matrix dimension length, and to allow easy maintenance of the user and server code.

For a simple use of the two-dimensional array, the alias name (new type definition) for this data type has been declared in the IDL. Declaring this alias name ensures that the same data type can be used across the client and server applications.

Parent topic: [eRPC example](#)

Creating the IDL file The created IDL file is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/`

The created IDL file contains the following code:

```
program erpc_matrix_multiply
/*! This const defines the matrix size. The value has to be the same as the
Matrix array dimension. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
const int32 matrix_size = 5;
/*! This is the matrix array type. The dimension has to be the same as the
matrix size const. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
type Matrix = int32[matrix_size][matrix_size];
interface MatrixMultiplyService {
erpcMatrixMultiply(in Matrix matrix1, in Matrix matrix2, out Matrix result_matrix) ->
void
}
```

Details:

- The IDL file starts with the program name (*erpc_matrix_multiply*), and this program name is used in the naming of all generated outputs.
- The declaration and definition of the constant variable named *matrix_size* follows next. The *matrix_size* variable is used for passing information about the length of matrix dimensions to the client/server user code.
- The alias name for the two-dimensional array type (*Matrix*) is declared.
- The interface group *MatrixMultiplyService* is located at the end of the IDL file. This interface group contains only one function declaration *erpcMatrixMultiply*.
- As shown above, the function’s declaration contains three parameters of *Matrix* type: *matrix1* and *matrix2* are input parameters, while *result_matrix* is the output parameter. Additionally, the returned data type is declared as *void*.

When writing the IDL file, the following order of items is recommended:

1. Program name at the top of the IDL file.
2. New data types and constants declarations.
3. Declarations of interfaces and functions at the end of the IDL file.

Parent topic: [eRPC example](#)

Using the eRPC generator tool | Windows OS | <MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x64
 | Linux OS | <MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86
 <MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86
 | | Mac OS | <MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Mac |

The files for the “Matrix multiply” example are pre-generated and already a part of the application projects. The following section describes how they have been created.

- The easiest way to create the shim code is to copy the erpcgen application to the same folder where the IDL file (*.erpc) is located; then run the following command:

```
erpcgen <IDL_file>.erpc
```

- In the “Matrix multiply” example, the command should look like:

```
erpcgen erpc_matrix_multiply.erpc
```

Additionally, another method to create the shim code is to execute the eRPC application using input commands:

- “-?”/”—help” – Shows supported commands.
- “-o <filePath>”/”—output<filePath>” – Sets the output directory.

For example,

```
<path_to_erpcgen>/erpcgen -o <path_to_output>  
<path_to_IDL>/<IDL_file_name>.erpc
```

For the “Matrix multiply” example, when the command is executed from the default erpcgen location, it looks like:

```
erpcgen -o
```

```
.././.././boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service  
.././.././boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/erpc_matrix_mu
```

In both cases, the following four files are generated into the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service folder:

- erpc_matrix_multiply.h
- erpc_matrix_multiply_client.cpp
- erpc_matrix_multiply_server.h
- erpc_matrix_multiply_server.cpp

For multiprocessor examples, the eRPC file and pre-generated files can be found in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_common/erpc_matrix_multiply/service folder.

For Linux OS users:

- Do not forget to set the permissions for the eRPC generator application.
- Run the application as ./erpcgen... instead of as erpcgen

Parent topic: [eRPC example](#)

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

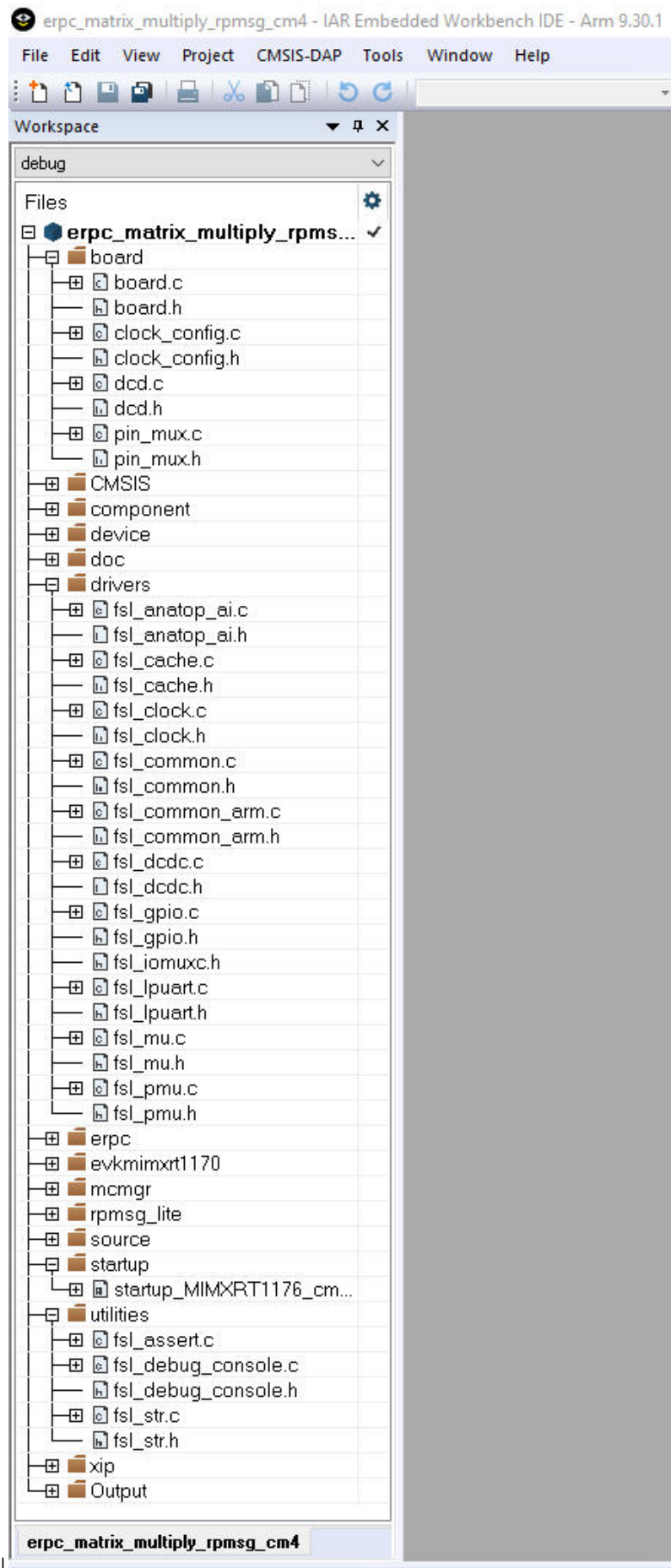
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

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

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



|

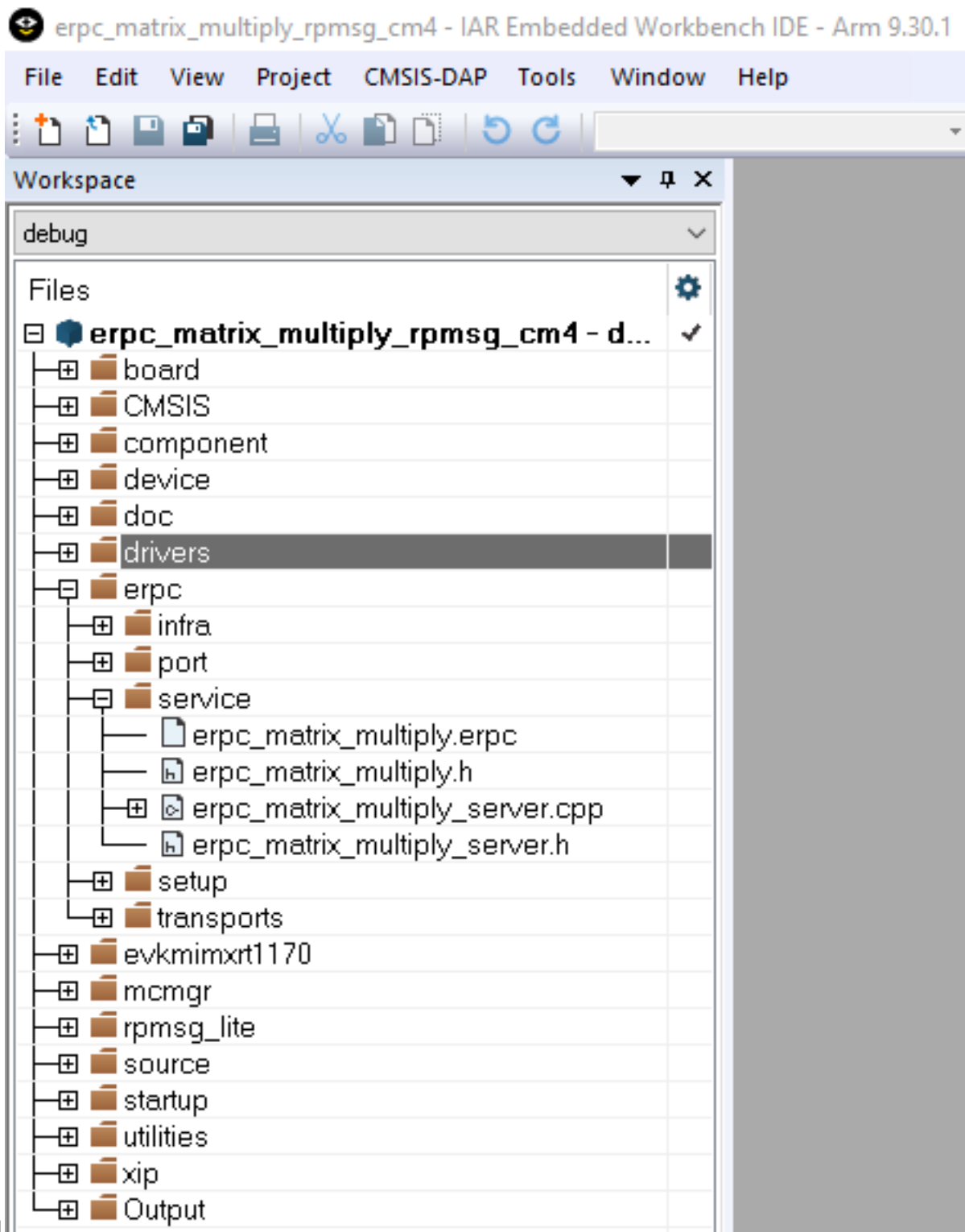
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/



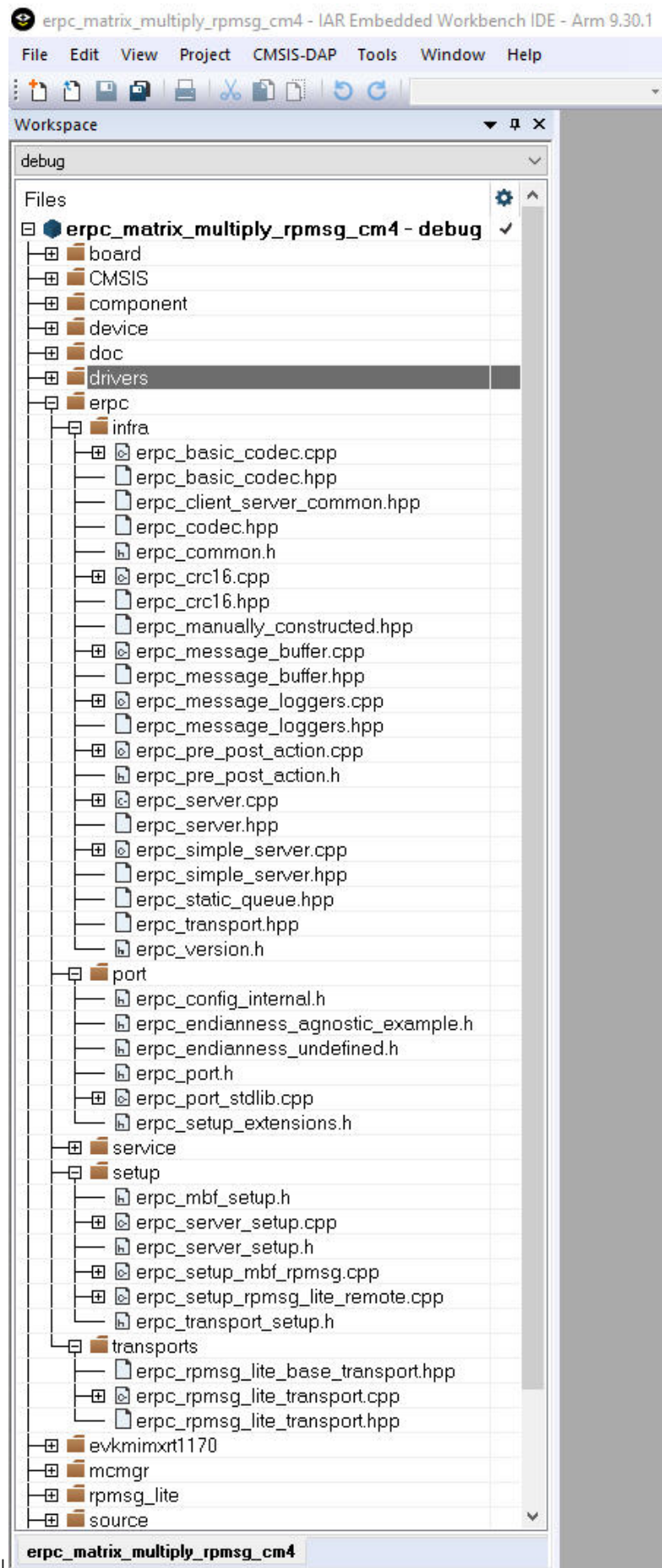
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

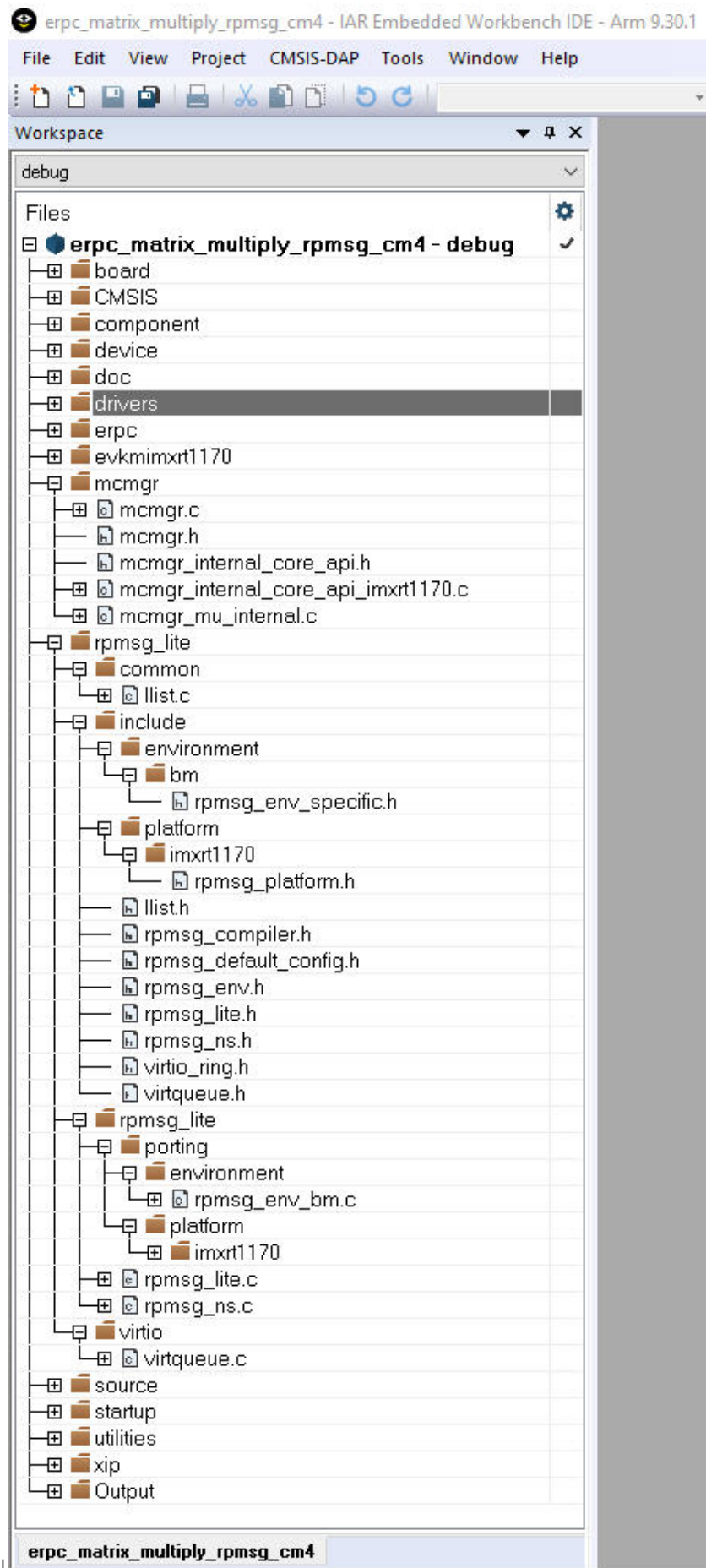
Parent topic: Multicore server application

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

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

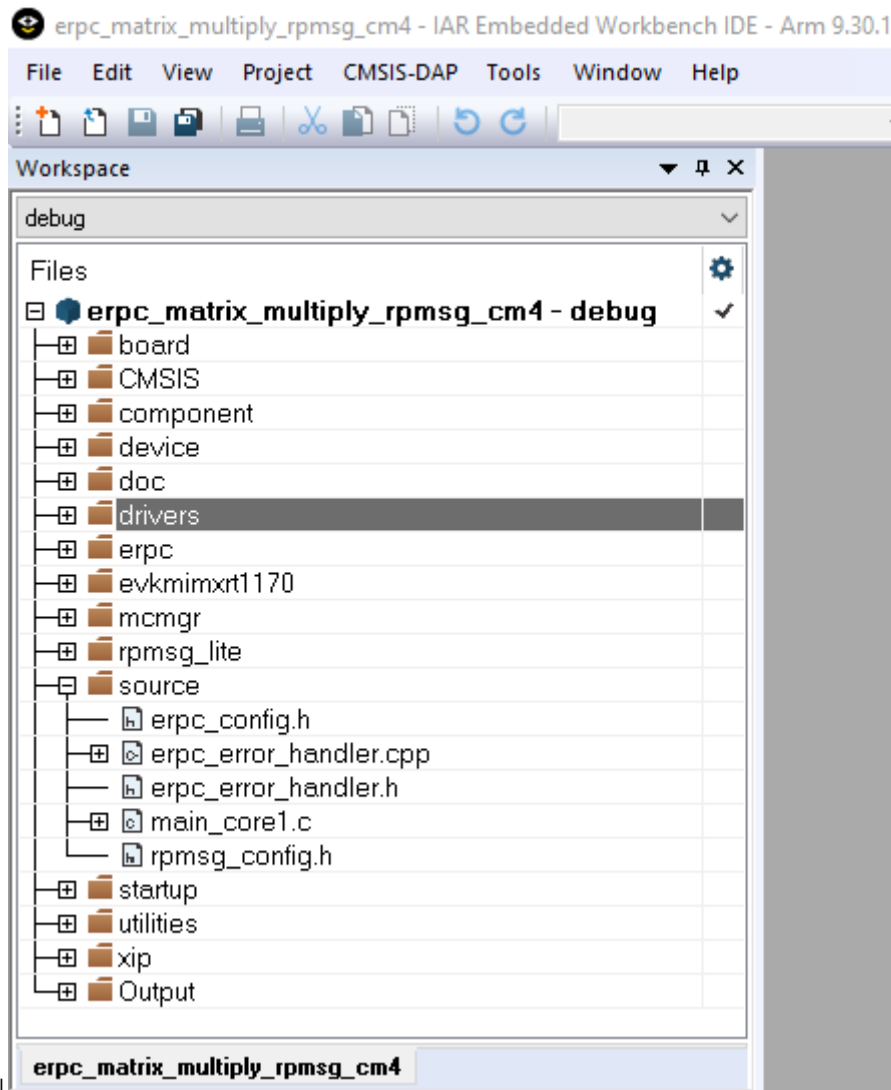
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg` folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

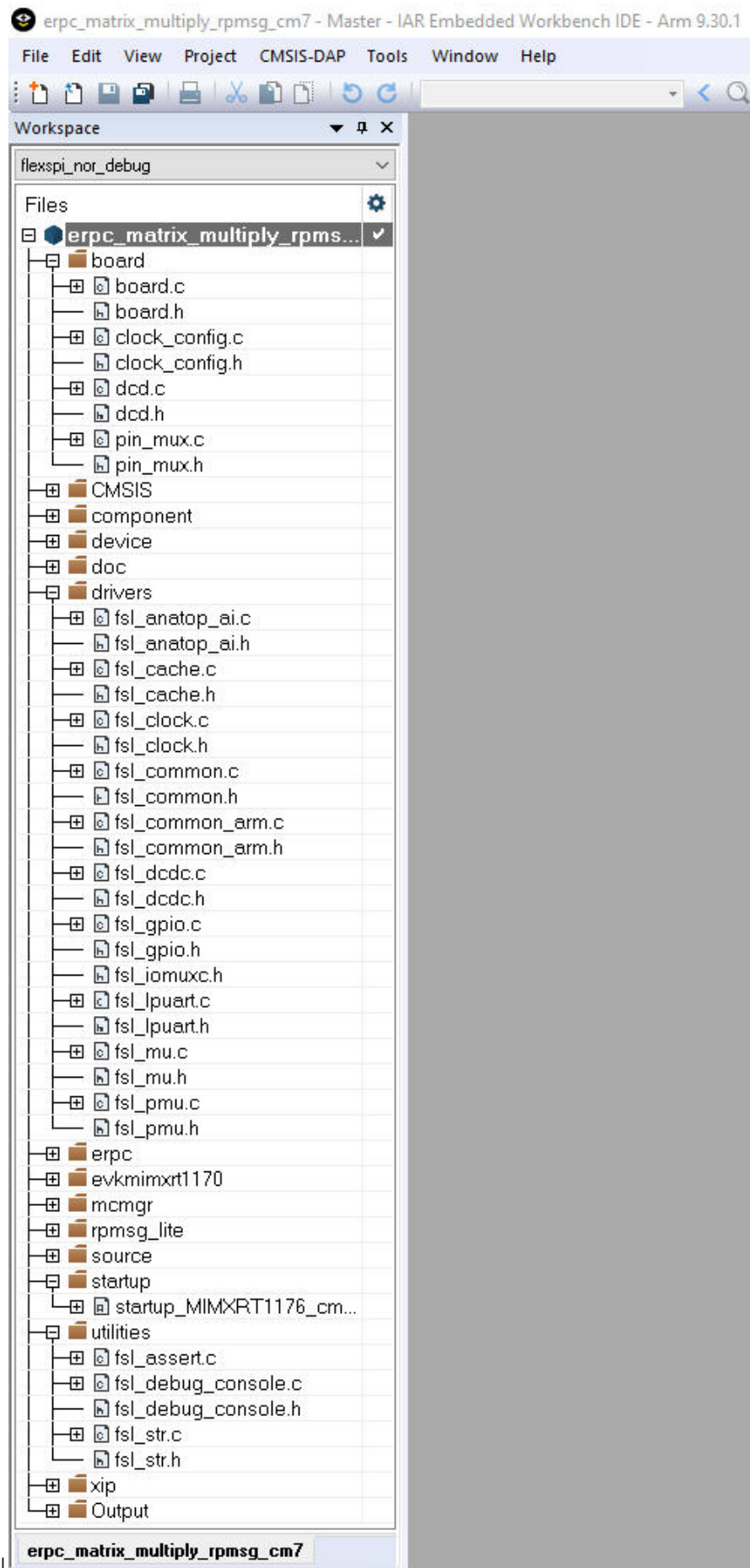
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar`

Project files for the eRPC client have the `_cm7` suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



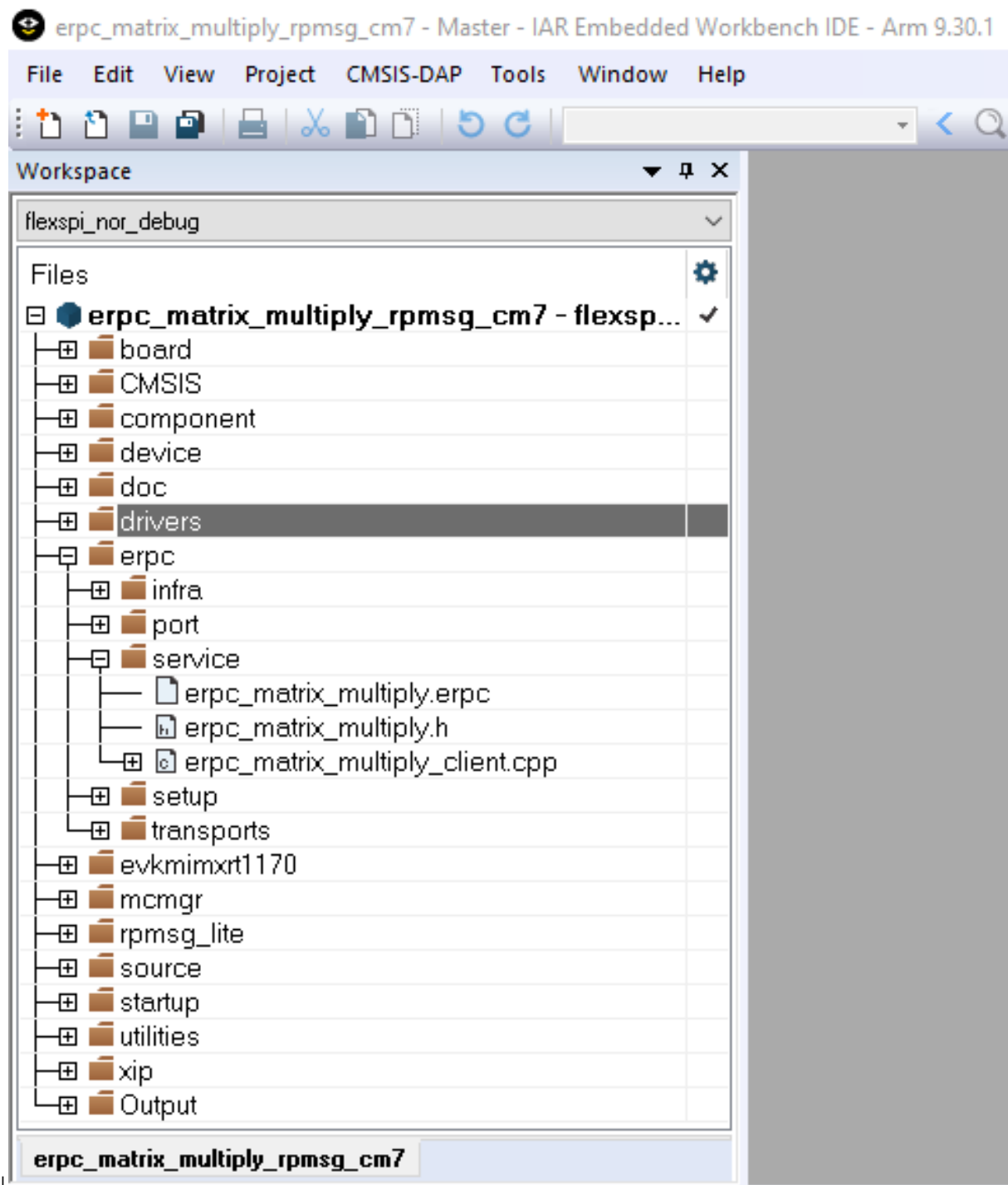
|

Parent topic:Multicore client application

Client-related generated files The client-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/ folder.



Parent topic: Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

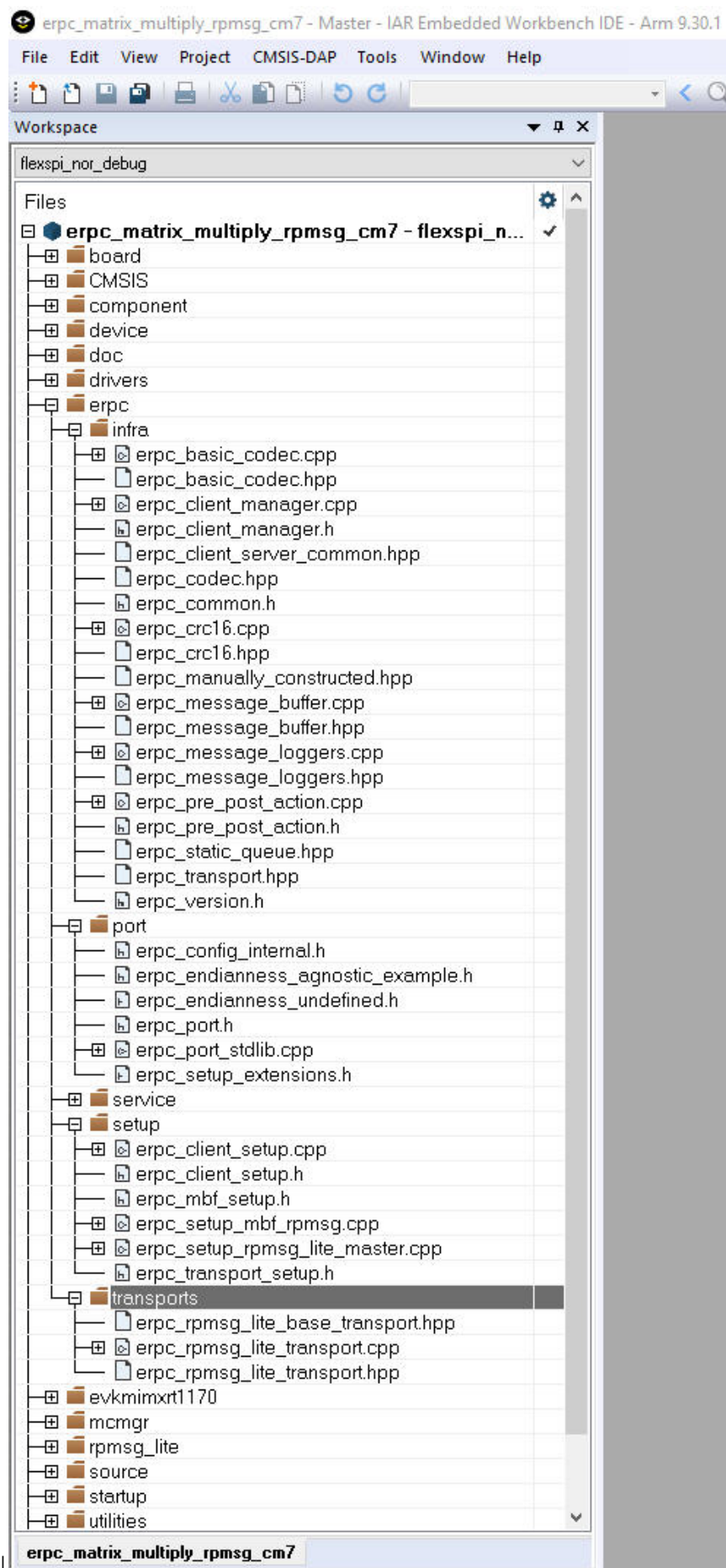
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

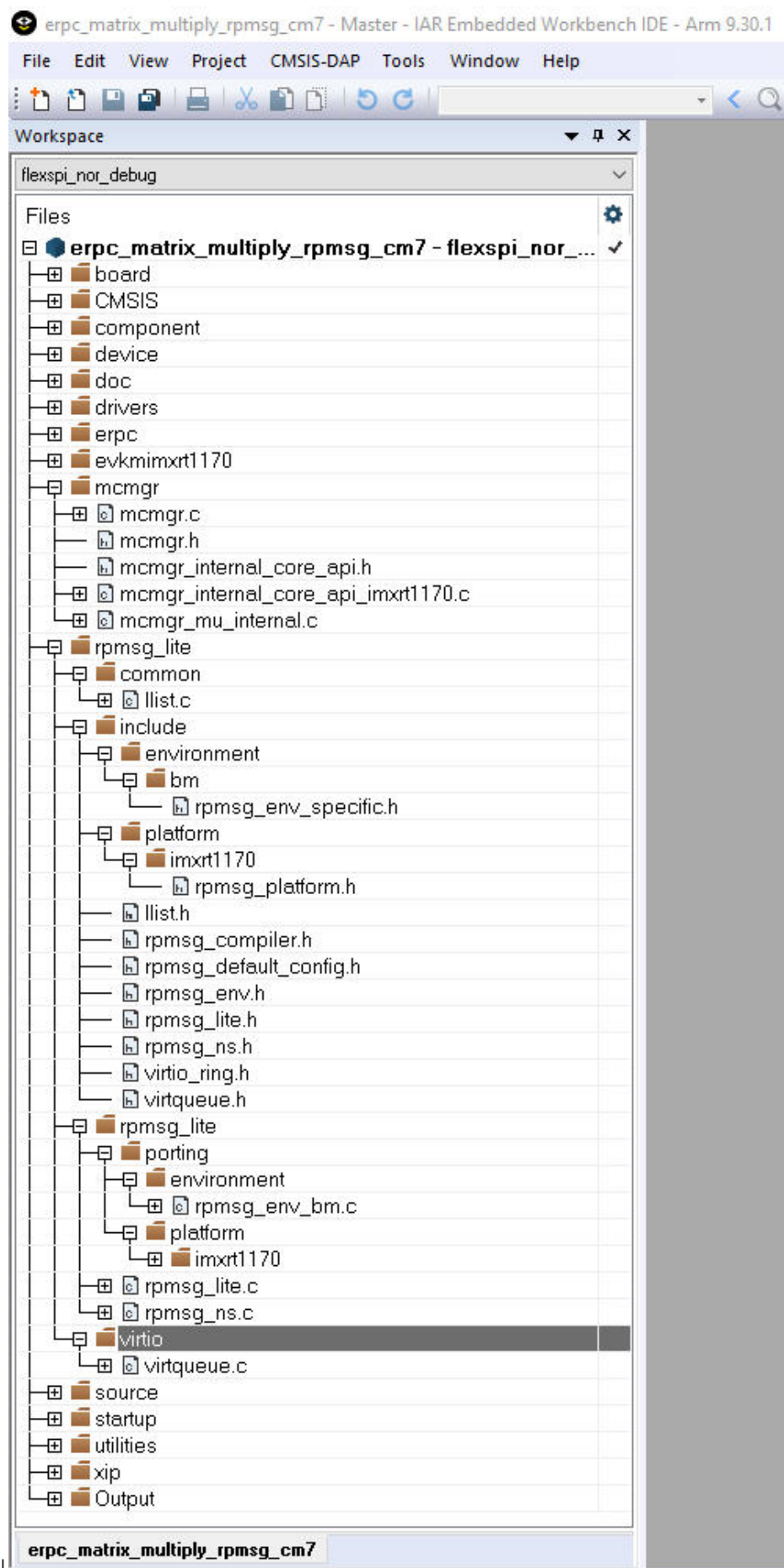
Parent topic:Multicore client application

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

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/



Parent topic:Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

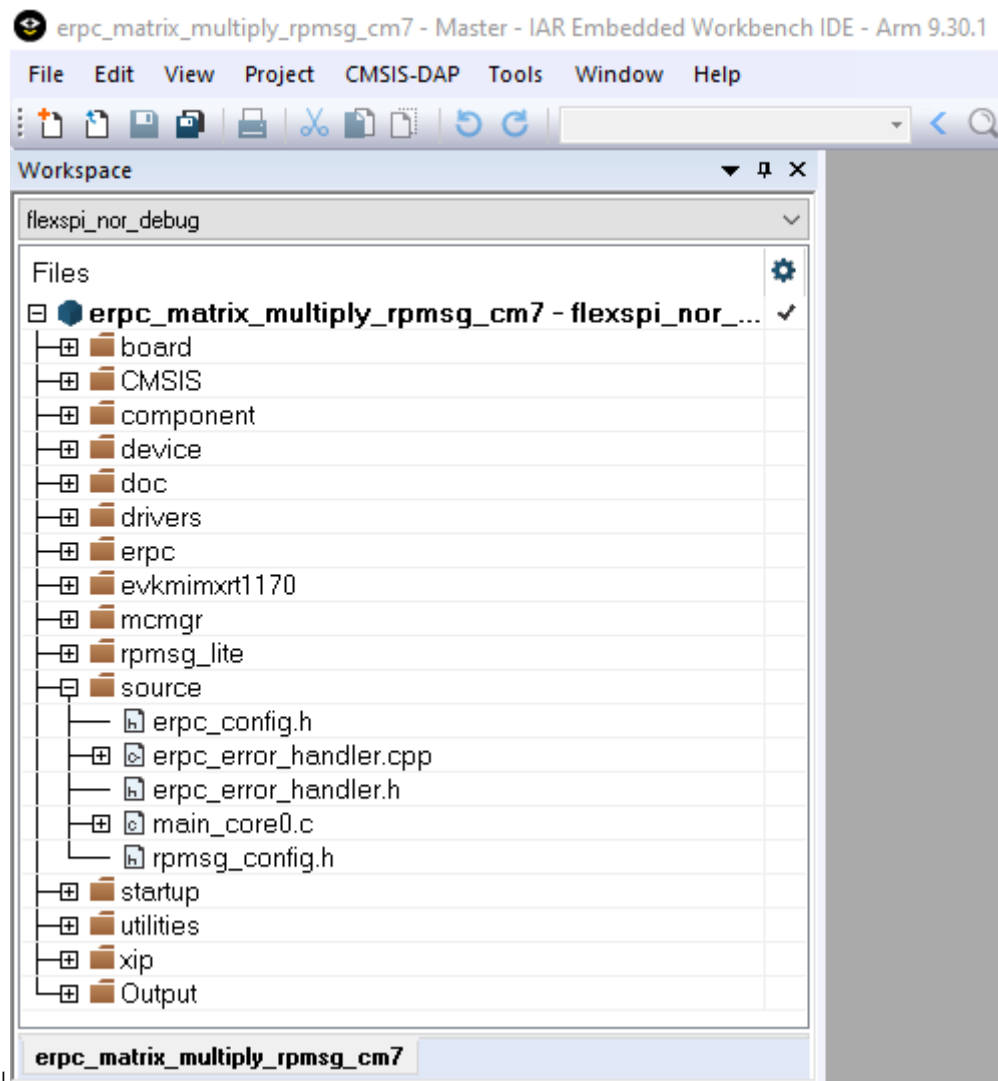
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic: Multicore client application

Parent topic: [Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RMPMsg-Lite). The RMPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.hpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.hpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

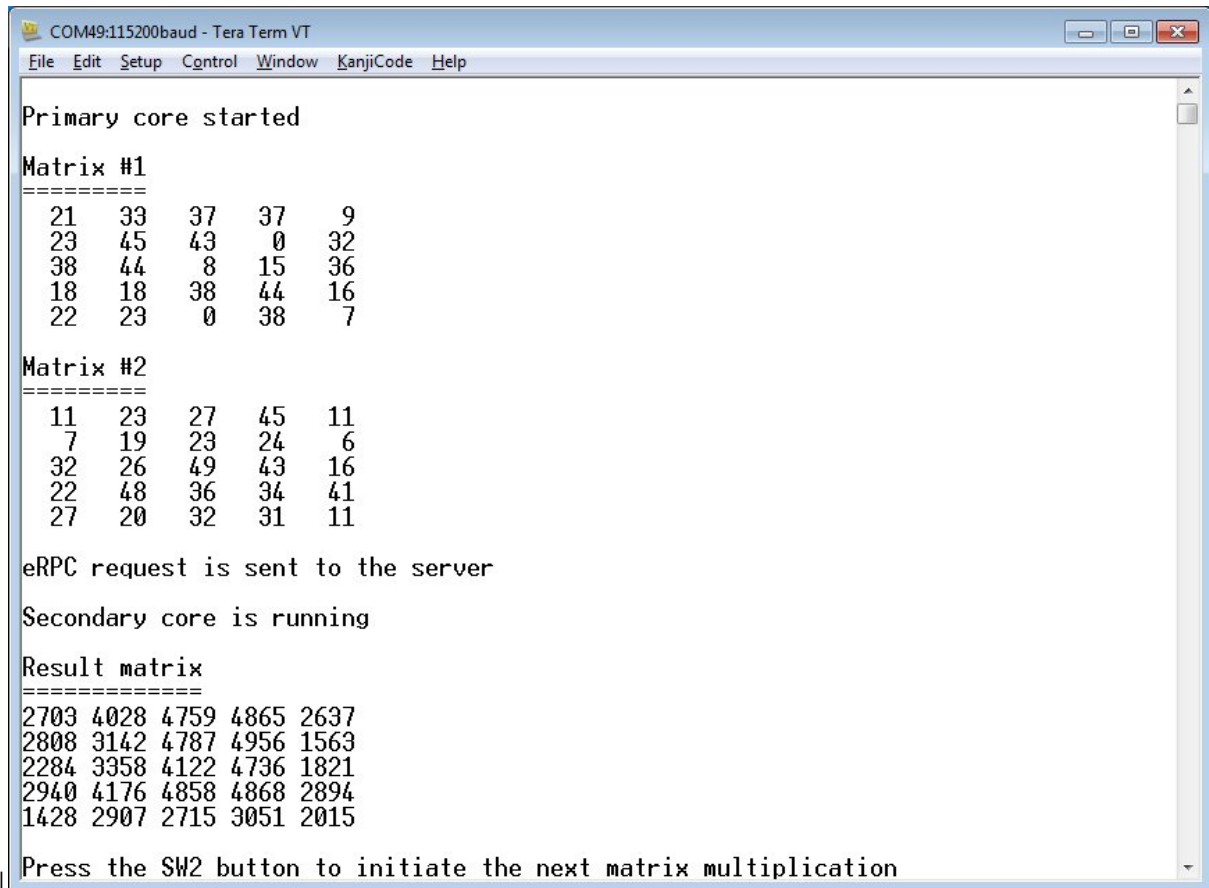
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

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

Primary core started

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

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

eRPC request is sent to the server

Secondary core is running

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

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

Other uses for an eRPC implementation The eRPC implementation is generic, and its use is not limited to just embedded applications. When creating an eRPC application outside the embedded world, the same principles apply. For example, this manual can be used to create an eRPC application for a PC running the Linux operating system. Based on the used type of transport medium, existing transport layers can be used, or new transport layers can be implemented.

For more information and erpc updates see the github.com/EmbeddedRPC.

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Changelog eRPC All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Added

Fixed

- Python code of the eRPC infrastructure was updated to match the proper python code style, add type annotations and improve readability.

1.14.0

Added

- Added Cmake/Kconfig support.
- Made java code jdk11 compliant, GitHub PR #432.
- Added imxrt1186 support into mu transport layer.
- erpcgen: Added assert for listType before usage, GitHub PR #406.

Fixed

- eRPC: Sources reformatted.
- erpc: Fixed typo in semaphore get (mutex -> semaphore), and write it can fail in case of timeout, GitHub PR #446.
- erpc: Free the arbitrated client token from client manager, GitHub PR #444.

- erpc: Fixed Makefile, install the erpc_simple_server header, GitHub PR #447.
- erpc_python: Fixed possible AttributeError and OSError on calling TCPTransport.close(), GitHub PR #438.
- Examples and tests consolidated.

1.13.0

Added

- erpc: Add BSD-3 license to endianness agnostic files, GitHub PR #417.
- eRPC: Add new Zephyr-related transports (zephyr_uart, zephyr_mbox).
- eRPC: Add new Zephyr-related examples.

Fixed

- eRPC,erpcgen: Fixing/improving markdown files, GitHub PR #395.
- eRPC: Fix Python client TCPTransports not being able to close, GitHub PR #390.
- eRPC,erpcgen: Align switch brackets, GitHub PR #396.
- erpc: Fix zephyr uart transport, GitHub PR #410.
- erpc: UART ZEPHYR Transport stop to work after a few transactions when using USB-CDC resolved, GitHub PR #420.

Removed

- eRPC,erpcgen: Remove cstbool library, GitHub PR #403.

1.12.0

Added

- eRPC: Add dynamic/static option for transport init, GitHub PR #361.
- eRPC,erpcgen: Winsock2 support, GitHub PR #365.
- eRPC,erpcgen: Feature/support multiple clients, GitHub PR #271.
- eRPC,erpcgen: Feature/buffer head - Framed transport header data stored in Message-Buffer, GitHub PR #378.
- eRPC,erpcgen: Add experimental Java support.

Fixed

- eRPC: Fix receive error value for spidev, GitHub PR #363.
- eRPC: UartTransport::init adaptation to changed driver.
- eRPC: Fix typo in assert, GitHub PR #371.
- eRPC,erpcgen: Move enums to enum classes, GitHub PR #379.
- eRPC: Fixed rpmsg tty transport to work with serial transport, GitHub PR #373.

1.11.0

Fixed

- eRPC: Makefiles update, GitHub PR #301.
- eRPC: Resolving warnings in Python, GitHub PR #325.
- eRPC: Python3.8 is not ready for usage of typing.Any type, GitHub PR #325.
- eRPC: Improved codec function to use reference instead of address, GitHub PR #324.
- eRPC: Fix NULL check for pending client creation, GitHub PR #341.
- eRPC: Replace sprintf with snprintf, GitHub PR #343.
- eRPC: Use MU_SendMsg blocking call in MU transport.
- eRPC: New LPSPI and LPI2C transport layers.
- eRPC: Freeing static objects, GitHub PR #353.
- eRPC: Fixed casting in deinit functions, GitHub PR #354.
- eRPC: Align LIBUSB_SIO.GetNumPorts API use with libusb_sio python module v. 2.1.11.
- erpcgen: Renamed temp variable to more generic one, GitHub PR #321.
- erpcgen: Add check that string read is not more than max length, GitHub PR #328.
- erpcgen: Move to g++ in pytest, GitHub PR #335.
- erpcgen: Use build=release for make, GitHub PR #334.
- erpcgen: Removed boost dependency, GitHub PR #346.
- erpcgen: Mingw support, GitHub PR #344.
- erpcgen: VS build update, GitHub PR #347.
- erpcgen: Modified name for common types macro scope, GitHub PR #337.
- erpcgen: Fixed memcpy for template, GitHub PR #352.
- eRPC,erpcgen: Change default build target to release + adding artefacts, GitHub PR #334.
- eRPC,erpcgen: Remove redundant includes, GitHub PR #338.
- eRPC,erpcgen: Many minor code improvements, GitHub PR #323.

1.10.0

Fixed

- eRPC: MU transport layer switched to blocking MU_SendMsg() API use.

1.10.0

Added

- eRPC: Add TCP_NODELAY option to python, GitHub PR #298.

Fixed

- eRPC: MUPort adaptation to new supported SoCs.
- eRPC: Simplifying CI with installing dependencies using shell script, GitHub PR #267.
- eRPC: Using event for waiting for sock connection in TCP python server, formatting python code, C specific includes, GitHub PR #269.
- eRPC: Endianness agnostic update, GitHub PR #276.
- eRPC: Assertion added for functions which are returning status on freeing memory, GitHub PR #277.
- eRPC: Fixed closing arbitrator server in unit tests, GitHub PR #293.
- eRPC: Makefile updated to reflect the correct header names, GitHub PR #295.
- eRPC: Compare value length to used length() in reading data from message buffer, GitHub PR #297.
- eRPC: Replace EXPECT_TRUE with EXPECT_EQ in unit tests, GitHub PR #318.
- eRPC: Adapt rpmsg_lite based transports to changed rpmsg_lite_wait_for_link_up() API parameters.
- eRPC, erpcgen: Better distinguish which file can and cannot be linked by C linker, GitHub PR #266.
- eRPC, erpcgen: Stop checking if pointer is NULL before sending it to the erpc_free function, GitHub PR #275.
- eRPC, erpcgen: Changed api to count with more interfaces, GitHub PR #304.
- erpcgen: Check before reading from heap the buffer boundaries, GitHub PR #287.
- erpcgen: Several fixes for tests and CI, GitHub PR #289.
- erpcgen: Refactoring erpcgen code, GitHub PR #302.
- erpcgen: Fixed assigning const value to enum, GitHub PR #309.
- erpcgen: Enable runTesttest_enumErrorCode_allDirection, serialize enums as int32 instead of uint32.

1.9.1

Fixed

- eRPC: Construct the USB CDC transport, rather than a client, GitHub PR #220.
- eRPC: Fix premature import of package, causing failure when attempting installation of Python library in a clean environment, GitHub PR #38, #226.
- eRPC: Improve python detection in make, GitHub PR #225.
- eRPC: Fix several warnings with deprecated call in pytest, GitHub PR #227.
- eRPC: Fix freeing union members when only default need be freed, GitHub PR #228.
- eRPC: Fix making test under Linux, GitHub PR #229.
- eRPC: Assert costumizing, GitHub PR #148.
- eRPC: Fix corrupt clientList bug in TransportArbitrator, GitHub PR #199.
- eRPC: Fix build issue when invoking g++ with -Wno-error=free-nonheap-object, GitHub PR #233.
- eRPC: Fix inout cases, GitHub PR #237.

- eRPC: Remove ERPC_PRE_POST_ACTION dependency on return type, GitHub PR #238.
- eRPC: Adding NULL to ptr when codec function failed, fixing memcpy when fail is present during deserialization, GitHub PR #253.
- eRPC: MessageBuffer usage improvement, GitHub PR #258.
- eRPC: Get rid of serial and enum34 dependency (enum34 is in python3 since 3.4 (from 2014)), GitHub PR #247.
- eRPC: Several MISRA violations addressed.
- eRPC: Fix timeout for Freertos semaphore, GitHub PR #251.
- eRPC: Use of rpmsg_lite_wait_for_link_up() in rpmsg_lite based transports, GitHub PR #223.
- eRPC: Fix codec nullptr dereferencing, GitHub PR #264.
- erpcgen: Fix two syntax errors in erpcgen Python output related to non-encapsulated unions, improved test for union, GitHub PR #206, #224.
- erpcgen: Fix serialization of list/binary types, GitHub PR #240.
- erpcgen: Fix empty list parsing, GitHub PR #72.
- erpcgen: Fix templates for malloc errors, GitHub PR #110.
- erpcgen: Get rid of encapsulated union declarations in global scale, improve enum usage in unions, GitHub PR #249, #250.
- erpcgen: Fix compile error:UniqueIdChecker.cpp:156:104:'sort' was not declared, GitHub PR #265.

1.9.0

Added

- eRPC: Allow used LIBUSB_SIO device index being specified from the Python command line argument.

Fixed

- eRPC: Improving template usage, GitHub PR #153.
- eRPC: run_clang_format.py cleanup, GitHub PR #177.
- eRPC: Build TCP transport setup code into liberpc, GitHub PR #179.
- eRPC: Fix multiple definitions of g_client error, GitHub PR #180.
- eRPC: Fix memset past end of buffer in erpc_setup_mbf_static.cpp, GitHub PR #184.
- eRPC: Fix deprecated error with newer pytest version, GitHub PR #203.
- eRPC, erpcgen: Static allocation support and usage of rpmsg static FreeRTOSs related API, GitHub PR #168, #169.
- erpcgen: Remove redundant module imports in erpcgen, GitHub PR #196.

1.8.1

Added

- eRPC: New i2c_slave_transport transport introduced.

Fixed

- eRPC: Fix misra erpc c, GitHub PR #158.
- eRPC: Allow conditional compilation of message_loggers and pre_post_action.
- eRPC: (D)SPI slave transports updated to avoid busy loops in rtos environments.
- erpcgen: Re-implement EnumMember::hasValue(), GitHub PR #159.
- erpcgen: Fixing several misra issues in shim code, erpcgen and unit tests updated, GitHub PR #156.
- erpcgen: Fix bison file, GitHub PR #156.

1.8.0

Added

- eRPC: Support win32 thread, GitHub PR #108.
- eRPC: Add mbed support for malloc() and free(), GitHub PR #92.
- eRPC: Introduced pre and post callbacks for eRPC call, GitHub PR #131.
- eRPC: Introduced new USB CDC transport.
- eRPC: Introduced new Linux spidev-based transport.
- eRPC: Added formatting extension for VSC, GitHub PR #134.
- erpcgen: Introduce ustring type for unsigned char and force cast to char*, GitHub PR #125.

Fixed

- eRPC: Update makefile.
- eRPC: Fixed warnings and error with using MessageLoggers, GitHub PR #127.
- eRPC: Extend error msg for python server service handle function, GitHub PR #132.
- eRPC: Update CMSIS UART transport layer to avoid busy loops in rtos environments, introduce semaphores.
- eRPC: SPI transport update to allow usage without handshaking GPIO.
- eRPC: Native _WIN32 erpc serial transport and threading.
- eRPC: Arbitrator deadlock fix, TCP transport updated, TCP setup functions introduced, GitHub PR #121.
- eRPC: Update of matrix_multiply.py example: Add -serial and -baud argument, GitHub PR #137.
- eRPC: Update of .clang-format, GitHub PR #140.
- eRPC: Update of erpc_framed_transport.cpp: return error if received message has zero length, GitHub PR #141.
- eRPC, erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136, #139.
- eRPC, erpcgen: Core re-formatted using Clang version 10.
- erpcgen: Enable deallocation in server shim code when callback/function pointer used as out parameter in IDL.
- erpcgen: Removed '\$' character from generated symbol name in '_\$union' suffix, GitHub PR #103.

- erpcgen: Resolved mismatch between C++ and Python for callback index type, GitHub PR #111.
- erpcgen: Python generator improvements, GitHub PR #100, #118.
- erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136.

1.7.4

Added

- eRPC: Support MU transport unit testing.
- eRPC: Adding mbed os support.

Fixed

- eRPC: Unit test code updated to handle service add and remove operations.
- eRPC: Several MISRA issues in rpmsg-based transports addressed.
- eRPC: Fixed Linux/TCP acceptance tests in release target.
- eRPC: Minor documentation updates, code formatting.
- erpcgen: Whitespace removed from C common header template.

1.7.3

Fixed

- eRPC: Improved the test_callbacks logic to be more understandable and to allow requested callback execution on the server side.
- eRPC: TransportArbitrator::prepareClientReceive modified to avoid incorrect return value type.
- eRPC: The ClientManager and the ArbitratedClientManager updated to avoid performing client requests when the previous serialization phase fails.
- erpcgen: Generate the shim code for destroy of statically allocated services.

1.7.2

Added

- eRPC: Add missing doxygen comments for transports.

Fixed

- eRPC: Improved support of const types.
- eRPC: Fixed Mac build.
- eRPC: Fixed serializing python list.
- eRPC: Documentation update.

1.7.1

Fixed

- eRPC: Fixed semaphore in static message buffer factory.
- erpcgen: Fixed MU received error flag.
- erpcgen: Fixed tcp transport.

1.7.0

Added

- eRPC: List names are based on their types. Names are more deterministic.
- eRPC: Service objects are as a default created as global static objects.
- eRPC: Added missing doxygen comments.
- eRPC: Added support for 64bit numbers.
- eRPC: Added support of program language specific annotations.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Generating crc value is optional.
- eRPC: Fixed CMSIS Uart driver. Removed dependency on KSDK.
- eRPC: Forbid users use reserved words.
- eRPC: Removed outByref for function parameters.
- eRPC: Optimized code style of callback functions.

1.6.0

Added

- eRPC: Added @nullable support for scalar types.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Improved eRPC nested calls.
- eRPC: Improved eRPC list length variable serialization.

1.5.0

Added

- eRPC: Added support for unions type non-wrapped by structure.
- eRPC: Added callbacks support.
- eRPC: Added support @external annotation for functions.
- eRPC: Added support @name annotation.
- eRPC: Added Messaging Unit transport layer.
- eRPC: Added RPMSG Lite RTOS TTY transport layer.
- eRPC: Added version verification and IDL version verification between eRPC code and eRPC generated shim code.
- eRPC: Added support of shared memory pointer.
- eRPC: Added annotation to forbid generating const keyword for function parameters.
- eRPC: Added python matrix multiply example.
- eRPC: Added nested call support.
- eRPC: Added struct member “byref” option support.
- eRPC: Added support of forward declarations of structures
- eRPC: Added Python RPMsg Multiendpoint kernel module support
- eRPC: Added eRPC sniffer tool

1.4.0

Added

- eRPC: New RPMsg-Lite Zero Copy (RPMsgZC) transport layer.

Fixed

- eRPC: win_flex_bison.zip for windows updated.
- eRPC: Use one codec (instead of inCodec outCodec).

[1.3.0]

Added

- eRPC: New annotation types introduced (@length, @max_length, ...).
- eRPC: Support for running both erpc client and erpc server on one side.
- eRPC: New transport layers for (LP)UART, (D)SPI.
- eRPC: Error handling support.

[1.2.0]

Added

- eRPC source directory organization changed.
- Many eRPC improvements.

[1.1.0]

Added

- Multicore SDK 1.1.0 ported to KSDK 2.0.0.

[1.0.0]

Added

- Initial Release

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

FreeRTOS kernel for MCUXpresso SDK Readme

FreeRTOS kernel for MCUXpresso SDK

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

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

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

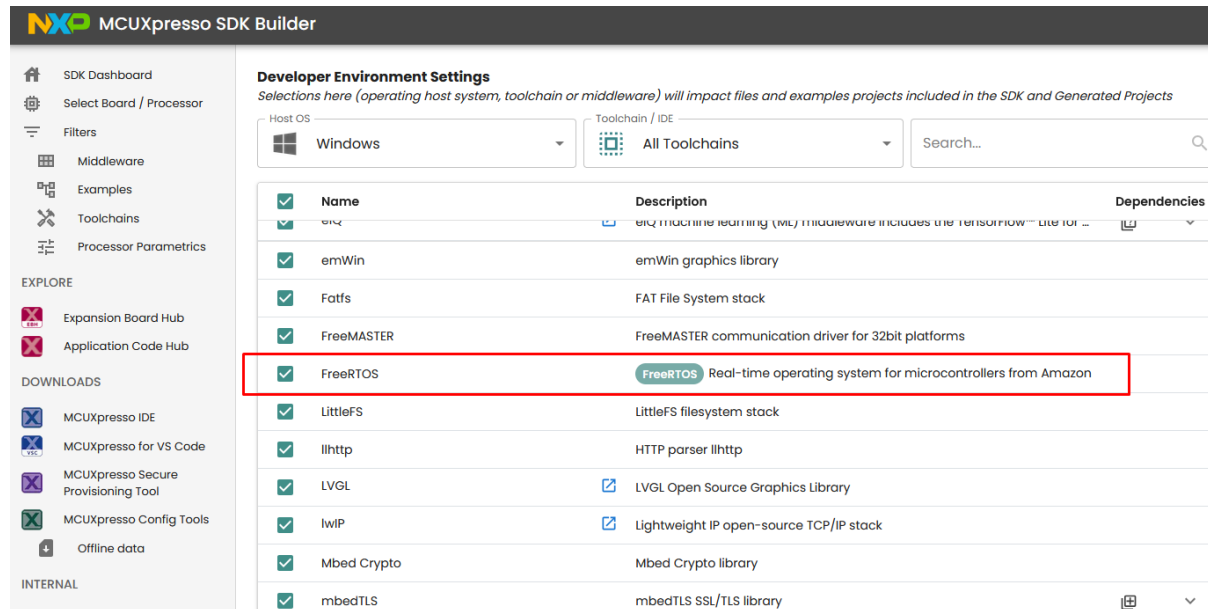
The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

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

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

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

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



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

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

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

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

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

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

FreeRTOS kernel for MCUXpresso SDK ChangeLog

Changelog FreeRTOS kernel for MCUXpresso SDK All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[Unreleased]**Added**

- Kconfig added CONFIG_FREERTOS_USE_CUSTOM_CONFIG_FRAGMENT config to optionally include custom FreeRTOSConfig fragment include file FreeRTOSConfig_frag.h. File must be provided by application.
- Added missing Kconfig option for configUSE_PICOLIBC_TLS.
- Add correct header files to build when configUSE_NEWLIB_REENTRANT and configUSE_PICOLIBC_TLS is selected in config.

[11.1.0_rev0]

- update amazon freertos version

[11.0.1_rev0]

- update amazon freertos version

[10.5.1_rev0]

- update amazon freertos version

[10.4.3_rev1]

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

[10.4.3_rev0]

- update amazon freertos version.

[10.4.3_rev0]

- update amazon freertos version.

[9.0.0_rev3]

- New features:
 - Tickless idle mode support for Cortex-A7. Add fsl_tickless_epit.c and fsl_tickless_generic.h in portable/IAR/ARM_CA9 folder.
 - Enabled float context saving in IAR for Cortex-A7. Added configUSE_TASK_FPU_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM_CA9 folder.
- Other changes:
 - Transformed ARM_CM core specific tickless low power support into generic form under freertos/Source/portable/low_power_tickless/.

[9.0.0_rev2]

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

[9.0.0_rev1]

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

[9.0.0_rev0]

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

[8.2.3]

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

FreeRTOS kernel Readme

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

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



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

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

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

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

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

To consume FreeRTOS-Kernel

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

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

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

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

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

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

– include/FreeRTOSConfig.h

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
INTERFACE
  include
)

target_compile_definitions(freertos_config
INTERFACE
  projCOVERAGE_TEST=0
)
```

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

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

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

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

FetchContent_MakeAvailable(freertos_kernel)
```

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

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

Consuming stand-alone - Cloning this repository

To clone using HTTPS:

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

Using SSH:

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

Repository structure

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

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

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

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

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

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

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

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

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

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

Readme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

return dnsStatus;
}

```

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

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

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

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

gcc can also produce an output file to be linked:

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

Building unit tests

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

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

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

Platform Prerequisites

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

Steps to build Unit Tests

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

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

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

MCUXpresso SDK: coreHTTP Client Library

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

coreHTTP Client Library

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

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

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

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

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

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

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

The HTTP client library can be built by either:

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

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

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

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

Building Unit Tests

Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

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

4.1.5 corejson

JSON parser.

Readme

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

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

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

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

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

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

Reference example

```

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

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

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

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

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

    return 0;
}

```

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

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

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

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

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

gcc can also produce an output file to be linked:

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

Building unit tests

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

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

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

Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

MCUXpresso SDK: coreMQTT Library

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

coreMQTT Client Library

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

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

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

coreMQTT v2.1.1 source code is part of the FreeRTOS 202210.01 LTS release.

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

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

Thus, the MQTT library can be built by either:

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

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

Format

The format of the username string with metrics is:

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

Where

- <Actual_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.
- <OS_Name> is the Operating System the application is running on (e.g. FreeRTOS)
- <OS_Version> is the version number of the Operating System (e.g. V10.4.3)
- <Hardware_Platform> is the Hardware Platform the application is running on (e.g. WinSim)
- <MQTT_Library_name> is the MQTT Client library being used (e.g. coreMQTT)
- <MQTT_Library_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual_Username = "iotuser", OS_Name = FreeRTOS, OS_Version = V10.4.3, Hardware_Platform_Name = WinSim, MQTT_Library_Name = coremqtt, MQTT_Library_version = 2.1.1. If username is not used, then "iotuser" can be removed.

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

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

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

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

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

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

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

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

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

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

Building Unit Tests

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

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

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

Platform Prerequisites

- Docker

or the following:

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

Steps to build Unit Tests

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

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

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

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

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

Platform	Location	Transport Interface Implementation
POSIX	AWS IoT Device SDK for Embedded C	POSIX sockets for TCP/IP and OpenSSL for TLS stack
FreeRTOS	FreeRTOS/FreeRTOS	FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack
FreeRTOS	FreeRTOS AWS Reference Integrations	Based on Secure Sockets Abstraction

Documentation

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

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

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

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

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

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

4.1.7 coremqtt-agent

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

Readme

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

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

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

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

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

To clone using HTTPS:

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

Using SSH:

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

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

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

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

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

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

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

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

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

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

Additional Considerations

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

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

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

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

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

Building Unit Tests

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

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

Unit Test Platform Prerequisites

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

Steps to build Unit Tests

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

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

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

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

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

Location
coreMQTT Agent Demos
FreeRTOS/FreeRTOS

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

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

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

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

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

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

4.1.8 corepkcs11

PKCS #11 key management library.

Readme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Build Prerequisites

Library Usage For building the library the following are required:

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

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

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

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

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

Coverage Measurement and Instrumentation The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.
- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.
- **gcov** binary corresponding to your chosen compiler
- **lcov** from the [Linux Test Project](#)
- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.
2. Run **cmake** to construct a build tree: `cmake -S test -B build`
 - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.
 - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.
3. Build the test binaries: `cmake --build ./build --target all`
4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.
5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

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

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

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

Reference examples The FreeRTOS-Labs repository contains demos using the PKCS #11 library [here](#) using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

Porting Guide Documentation for porting corePKCS11 to a new platform can be found on the AWS [docs](#) web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

Related Example Implementations These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's [Platform Security Architecture](#).
- Microchip's [cryptoauthlib](#).
- Infineon's [Optiga Trust X](#).

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

Note that the latest included version of corePKCS11 may differ across repositories.

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

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

Security See *CONTRIBUTING* for more information.

License This library is licensed under the MIT-0 License. See the LICENSE file.

4.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme

MCUXpresso SDK: FreeRTOS-Plus-TCP Library This repository is a fork of FreeRTOS-Plus-TCP library (<https://github.com/FreeRTOS/freertos-plus-tcp>)(4.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

Introduction This branch contains unified IPv4 and IPv6 functionalities. Refer to the Getting started Guide (found [here](#)) for more details.

FreeRTOS-Plus-TCP Library FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the [MISRA coding standard](#). Any deviations from the MISRA C:2012 guidelines are documented under [MISRA Deviations](#). The library is validated for memory safety and data structure invariance through the [CBMC automated reasoning tool](#) for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

Getting started The easiest way to use the 4.0.0 version of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found [here](#)) Another way is to start with the pre-configured demo application project (found in [this directory](#)). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the [Documentation](#), and [API Reference](#).

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#). Please also refer to [FAQ](#) for frequently asked questions.

Also see the [Submitting a bugs/feature request](#) section of CONTRIBUTING.md for more details.

Note: All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

Upgrading to V3.0.0 and V3.1.0 In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a `source` directory. This change requires modification of any existing project(s) to include the modified source files and directories. There are examples on how to use the new files and directory structure. For an example based on the Xilinx Zynq-7000, use the code in this [branch](#) and follow these [instructions](#) to build and run the demo.

FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 202210.00 LTS release.

Generating pre V3.0.0 folder structure for backward compatibility: If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to [this](#).

Note: After running the script, while the `.c` files will have same names as the pre V3.0.0 source, the files in the `include` directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from [here](#).

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing `python --version`.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory that was created using the *Cloning this repository* step above. And then run `python <Path/to/the/script>/GenerateOriginalFiles.py`.

To consume FreeRTOS+TCP

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

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

```
FetchContent_Declare( freertos_plus_tcp
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
  GIT_TAG        master #Note: Best practice to use specific git-hash or tagged version
  GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

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


```

set( FREERTOS_PLUS_FAT_DEV_SUPPORT OFF CACHE BOOL "" FORCE)
# Select the native compile PORT
set( FREERTOS_PLUS_FAT_PORT "POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  # Eg. Zynq 2019_3 version of port
  set(FREERTOS_PLUS_FAT_PORT "ZYNQ_2019_3" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)

```

Consuming stand-alone This repository uses [Git Submodules](#) to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```

git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Using SSH:

```

git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Porting The porting guide is available on [this page](#).

Repository structure This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
 - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
 - This directory contains all the tests (unit tests and CBMC) and the dependencies ([FreeRTOS-Kernel/Litani-port](#)) the tests require.
- source/portable
 - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
 - The include directory has all the ‘core’ header files of FreeRTOS-Plus-TCP source.
- source
 - This directory contains all the [.c] source files.

Note At this time it is recommended to use `BufferAllocation_2.c` in which case it is essential to use the `heap_4.c` memory allocation scheme. See [memory management](#).

Kernel sources The FreeRTOS Kernel Source is in [FreeRTOS/FreeRTOS-Kernel repository](#), and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at `./test/FreeRTOS-Kernel` directory.

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

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

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