



MCUXpresso SDK Documentation

Release 25.12.00-pvw2



NXP
Nov 14, 2025



Table of contents

1	MCIMX93-EVK	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	MCUXSDKMCIMX93EVKGSUG	3
1.3	Getting Started with MCUXpresso SDK GitHub	28
1.3.1	Getting Started with MCUXpresso SDK Repository	28
1.4	Release Notes	41
1.4.1	MCUXpresso SDK Release Notes	41
1.5	ChangeLog	46
1.5.1	MCUXpresso SDK Changelog	46
1.6	Driver API Reference Manual	146
1.7	Middleware Documentation	146
1.7.1	Multicore	146
1.7.2	FreeMASTER	146
1.7.3	FreeRTOS	146
1.7.4	lwIP	146
2	MIMX9352	147
2.1	eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	147
2.2	eDMA core Driver	172
2.3	eDMA soc Driver	179
2.4	ENET: Ethernet MAC Driver	179
2.5	EQOS-TSN: Ethernet QoS with TSN Driver	206
2.6	Enet_qos_qos	206
2.7	FGPIO Driver	246
2.8	FlexCAN: Flex Controller Area Network Driver	246
2.9	FlexCAN Driver	246
2.10	FlexCAN eDMA Driver	269
2.11	FlexIO: FlexIO Driver	271
2.12	FlexIO Driver	271
2.13	FlexIO eDMA I2S Driver	284
2.14	FlexIO eDMA SPI Driver	288
2.15	FlexIO eDMA UART Driver	291
2.16	FlexIO I2C Master Driver	294
2.17	FlexIO I2S Driver	303
2.18	FlexIO SPI Driver	313
2.19	FlexIO UART Driver	326
2.20	FLEXSPI: Flexible Serial Peripheral Interface Driver	336
2.21	FLEXSPI eDMA Driver	353
2.22	I3C: I3C Driver	356
2.23	I3C Common Driver	358
2.24	I3C Master Driver	361
2.25	I3C Master DMA Driver	387
2.26	I3C Slave Driver	390
2.27	I3C Slave DMA Driver	403
2.28	Iomuxc_driver	406

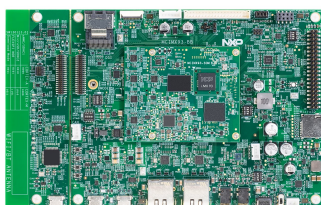
2.29	ISI: Image Sensing Interface	426
2.30	Common Driver	441
2.31	LCDIFv3: LCD Interface v3	453
2.32	Lin_lpuart_driver	461
2.33	LPI2C: Low Power Inter-Integrated Circuit Driver	468
2.34	LPI2C Master Driver	470
2.35	LPI2C Master DMA Driver	484
2.36	LPI2C Slave Driver	486
2.37	LPIT: Low-Power Interrupt Timer	497
2.38	LPSPi: Low Power Serial Peripheral Interface	503
2.39	LPSPi Peripheral driver	503
2.40	LPSPi eDMA Driver	525
2.41	LPTMR: Low-Power Timer	532
2.42	LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver	538
2.43	LPUART Driver	538
2.44	LPUART eDMA Driver	554
2.45	MCM: Miscellaneous Control Module	557
2.46	Mipi_dsi	561
2.47	MIPI_DSI: MIPI DSI Host Controller	578
2.48	MU: Messaging Unit	578
2.49	OTFAD: On The Fly AES-128 Decryption Driver	593
2.50	PDM: Microphone Interface	596
2.51	PDM Driver	596
2.52	PDM EDMA Driver	615
2.53	PXP: Pixel Pipeline	619
2.54	RGPIO: Rapid General-Purpose Input/Output Driver	644
2.55	RGPIO Driver	645
2.56	SAI: Serial Audio Interface	647
2.57	SAI Driver	647
2.58	SAI EDMA Driver	670
2.59	SEMA42: Hardware Semaphores Driver	677
2.60	SPDIF: Sony/Philips Digital Interface	681
2.61	SPDIF eDMA Driver	694
2.62	TMU: Thermal Management Unit Driver	697
2.63	TPM: Timer PWM Module	703
2.64	TRDC: Trusted Resource Domain Controller	715
2.65	Trdc_core	716
2.66	TRGMUX: Trigger Mux Driver	729
2.67	TSTMR: Timestamp Timer Driver	731
2.68	USDHC: Ultra Secured Digital Host Controller Driver	731
2.69	WDOG32: 32-bit Watchdog Timer	759
2.70	CACHE: CACHE Memory Controller	765
3	Middleware	769
3.1	Motor Control	769
3.1.1	FreeMASTER	769
4	RTOS	807
4.1	FreeRTOS	807
4.1.1	FreeRTOS kernel	807
4.1.2	FreeRTOS drivers	807
4.1.3	backoffalgorithm	807
4.1.4	corehttp	807
4.1.5	corejson	807
4.1.6	coremqtt	808
4.1.7	corepkcs11	808
4.1.8	freertos-plus-tcp	808

This documentation contains information specific to the mcimx93evk board.

Chapter 1

MCIMX93-EVK

1.1 Overview



MCU device and part on board is shown below:

- Device: MIMX9352
- PartNumber: MIMX9352DVVXM

1.2 Getting Started with MCUXpresso SDK Package

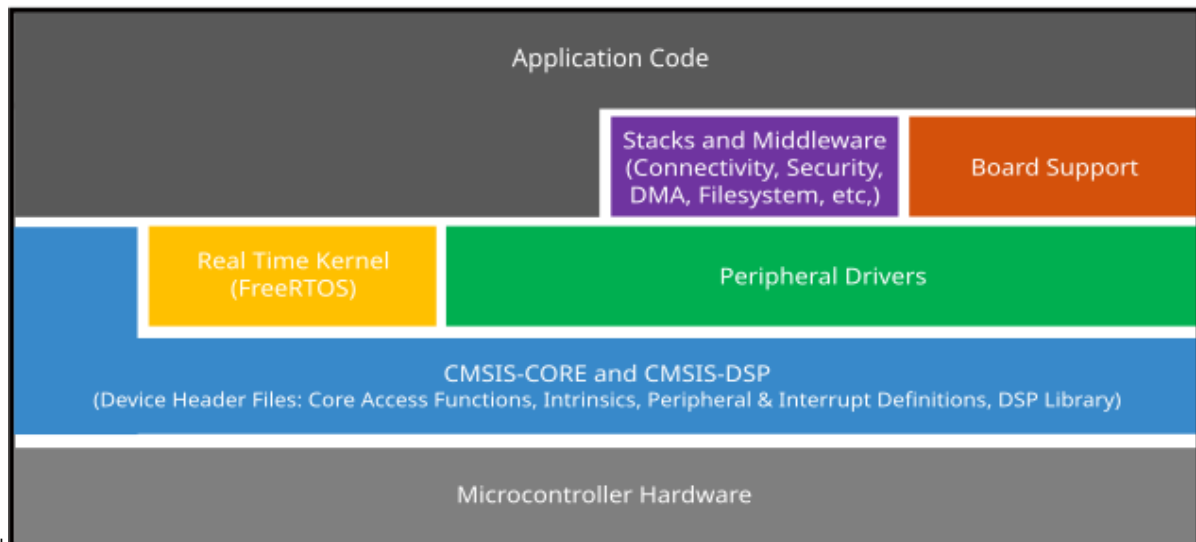
1.2.1 MCUXSDKMCIMX93EVKGSUG

Overview

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

For supported toolchain versions, see *MCUXpresso SDK Release Notes for MCIMX93-EVK (document MCUXSDKMCIMX93EVKRN)*.

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

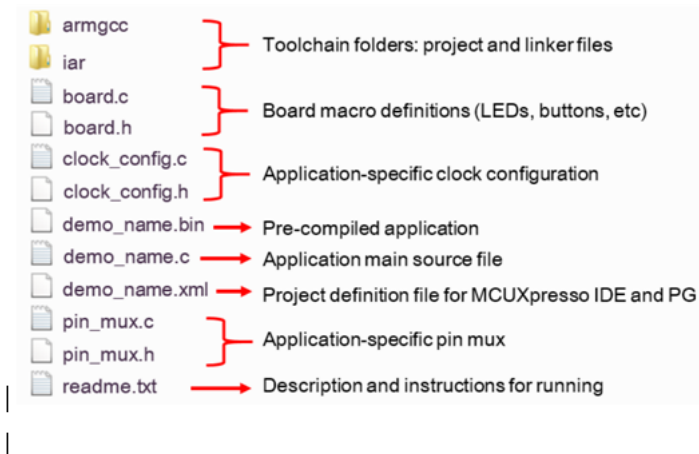
MCUXpresso SDK board support provides example applications for NXP development and evaluation boards for Arm Cortex-M cores. Board support packages are found inside of the top level boards folder, and each supported board has its own folder (MCUXpresso SDK package can support multiple boards). Within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

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

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



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

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

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

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

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

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

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

Toolchain introduction

The MCUXpresso SDK release for i.MX 93 includes the build system to be used with some toolchains. In this chapter, the toolchain support is presented and detailed.

Compiler/Debugger The MCUXpresso SDK i.MX 93 release supports building and debugging with the toolchains listed in Table 1.

The user 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 debugger probe and supports the device to attach, debug, and download.

Com- piler/Debugger	Supported host OS	Debug probe	Tool website
ArmGCC/J-Link GDB server	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 93, the patch for IAR and Segger J-Link should be installed. The patch named `iar_segger_support_patch_imx93.zip` can be used with MCUXpresso SDK. Contact the NXP representative for the zip package. For additional information about patch installation, see the *readme.txt* in the patch.

Parent topic: [Toolchain introduction](#)

Run a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the i.MX 93 EVK hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

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

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

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

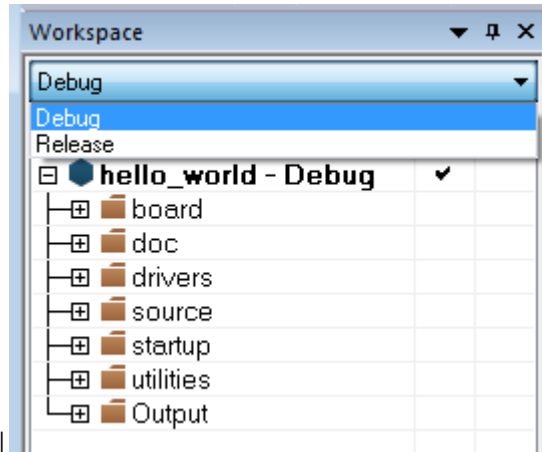
Using the i.MX 93 EVK hardware platform as an example, the `hello_world` workspace is located in:

```
<install_dir>/boards/mcimx93evk/demo_apps/hello_world/iar/hello_world.eww
```

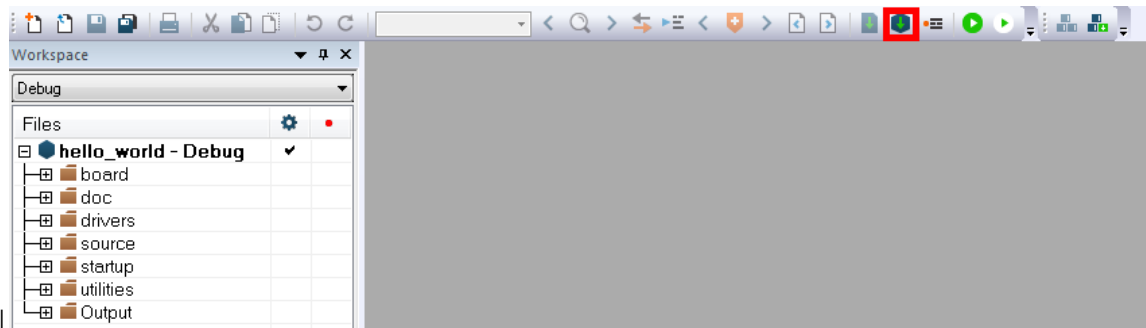
Other example applications may have additional folders in their path.

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

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



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

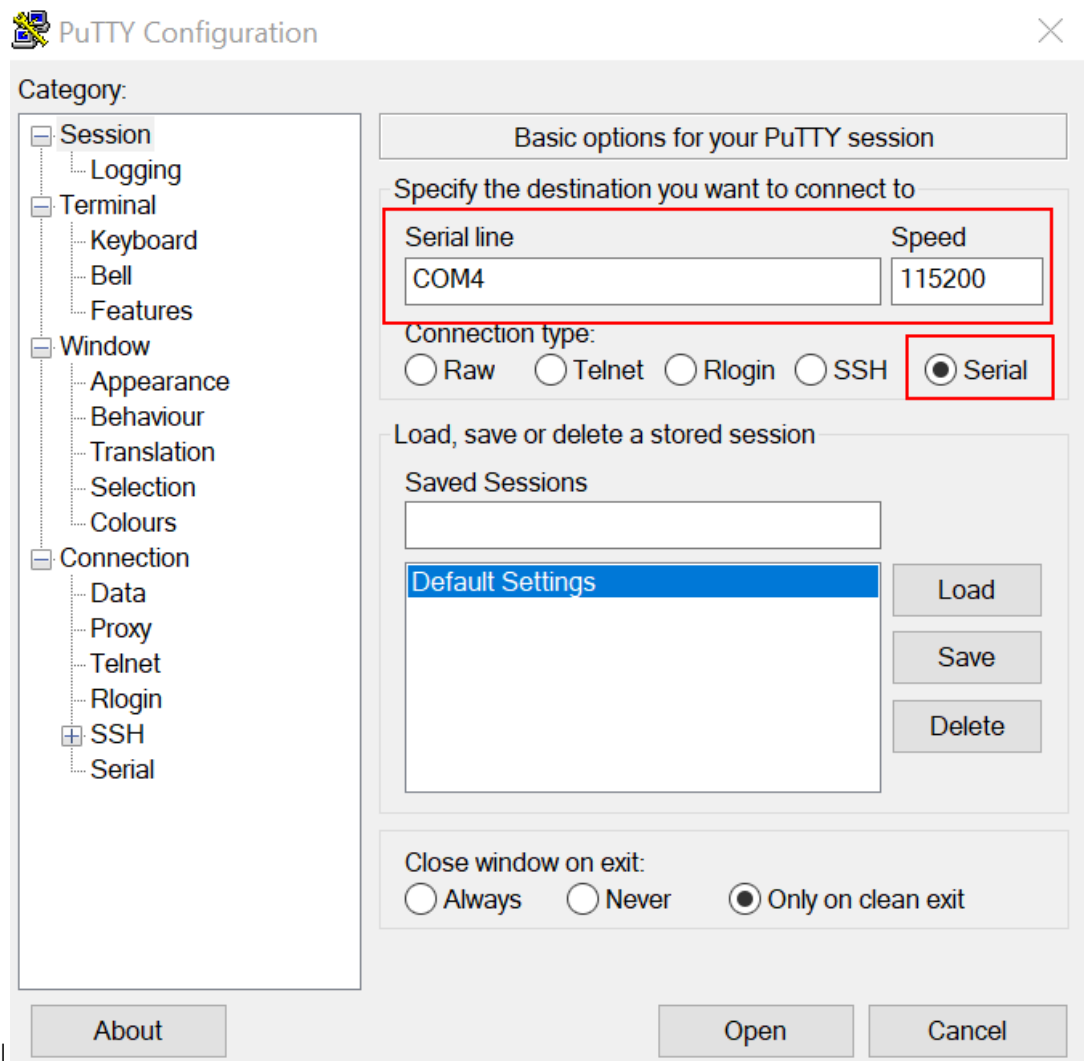


4. The build completes without errors.

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

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

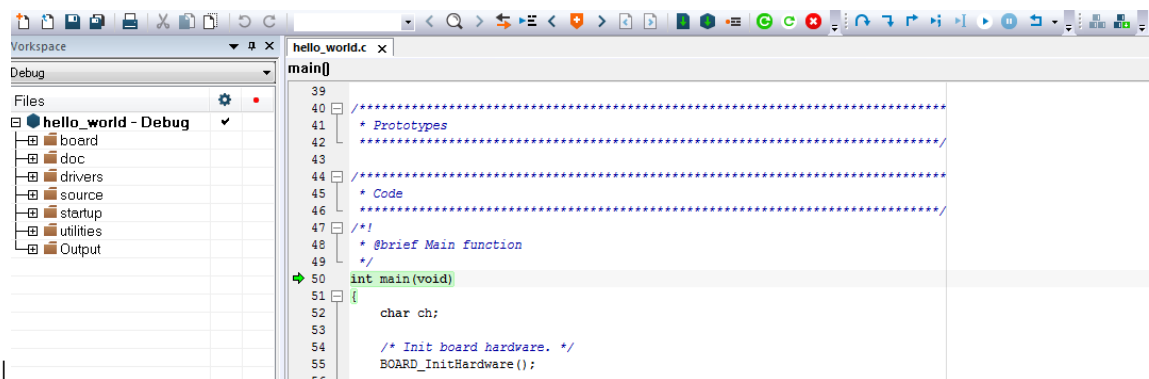
1. This board supports the J-Link PLUS debug probe. Before using it, install SEGGER J-Link software, which can be downloaded from <http://www.segger.com/downloads/jlink/>.
2. Connect the development platform to your PC via USB cable between the DBG USB connector (J1401) and the PC USB connector.
3. Connect 12 V ~ 20 V power supply and J-Link Plus to the device.
4. Switch SW1301[3:0] to the M core boot and ensure that the image is not available on the boot source. For example, 0b1010 for MicroSD boot. Keep the SD slot empty.
5. Open the terminal application on the PC, such as PuTTY or TeraTerm, connect to the debug COM port, see [How to determine COM port](#), and configure the terminal with these settings:
 1. 115,200 baud rate
 2. No parity
 3. 8 data bits
 4. 1 stop bit



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



7. The application then downloads to the target and automatically runs to the `main()` function.



8. Run the code by clicking **Go** to start the application.



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



****Note:**** If the software is already running on the M core, the debugger loading image into TCM may get
 ↳ HardFault or a data verification issue. NXP recommends you to follow the steps above to use the debugger.
 ↳ Repowering the board is required to restart the debugger.

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

Run a demo using Arm GCC

This section describes the steps to configure the command line Arm GCC tools to build, run, and debug demo applications and necessary driver libraries provided in the MCUXpresso SDK. The `hello_world` demo application targeted for i.MX 93 is used as an example, though these steps can be applied to any board, demo or example application in the MCUXpresso SDK.

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

Note: See [Host setup](#) for Linux OS 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 tool chain installation path. For this example, the path is:

```
$ export ARMGCC_DIR=/work/platforms/tmp/gcc-arm-none-eabi-9-2019-q4-major
```

```
$ export PATH= $PATH:/work/platforms/tmp/gcc-arm-none-eabi-9-2019-q4-major
```

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/mcimx93evk/ demo_apps/hello_world/armgcc.

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

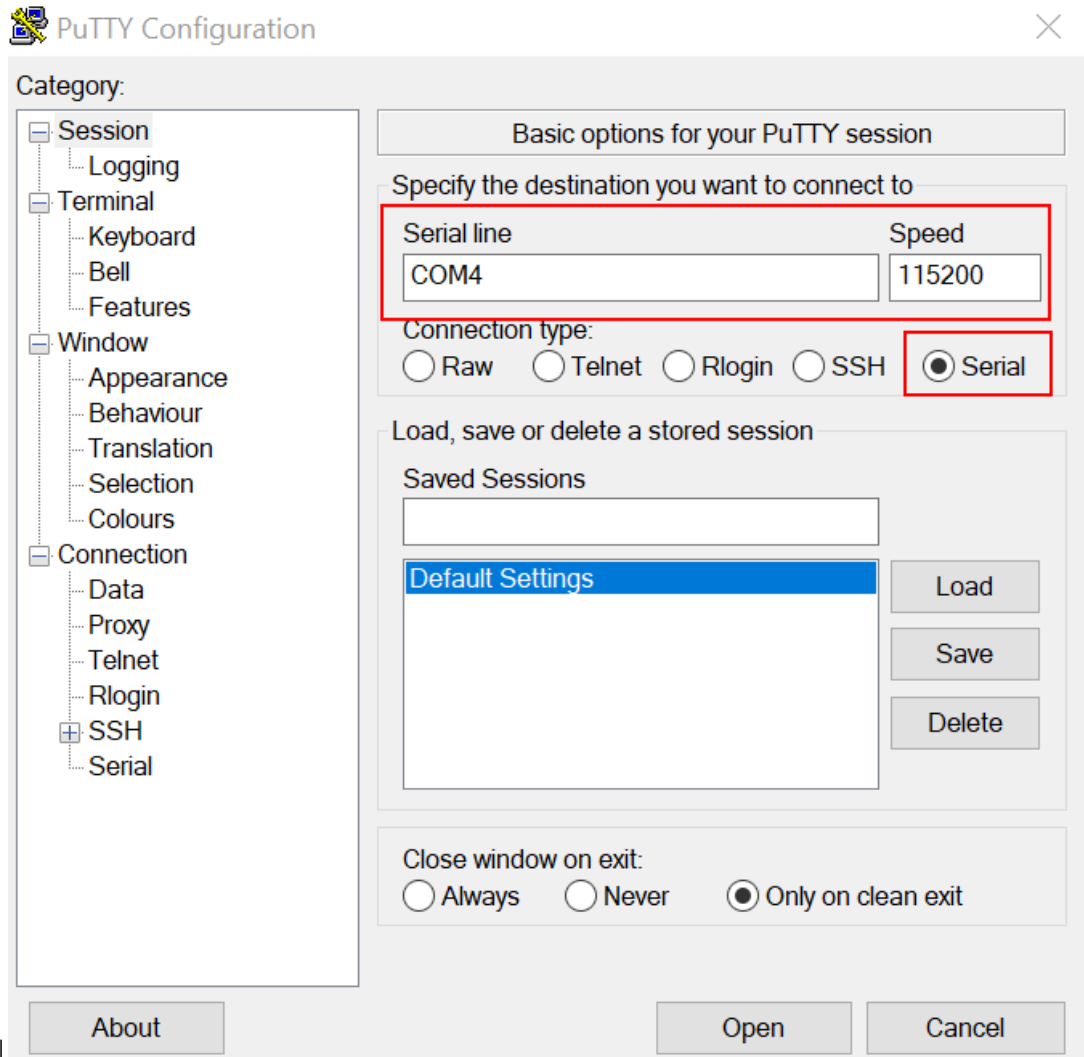
```
$ ./build_debug.sh
-- TOOLCHAIN_DIR: /work/platforms/tmp/gcc-arm-none-eabi-9-2019-q4-major
-- BUILD_TYPE: debug
-- TOOLCHAIN_DIR: /work/platforms/tmp/gcc-arm-none-eabi-9-2019-q4-major
-- BUILD_TYPE: debug
-- The ASM compiler identification is GNU
-- Found assembler: /work/platforms/tmp/gcc-arm-none-eabi-8-2019-q3-update/bin/arm-none-eabi-gcc
-- Configuring done
-- Generating done
-- Build files have been written to:
/work/platforms/tmp/nxp/SDK\2.12.0\MCIMX93\EVK/boards/mcimx93evk/demo_apps/hello\
↪_world/armgcc/demo_apps/hello_world/armgcc
Scanning dependencies of target hello_world.elf
\[ 6%\] Building C object CMakeFiles/hello_world.elf.dir/work/platforms/tmp/nxp/SDK\2.12.0\
↪MCIMX93\EVK/boards/mcimx93evk/demo_apps/hello_world/hello_world.c.obj
< -- skipping lines -- >
[100%] Linking C executable debug/hello_world.elf
[100%] Built target hello_world.elf
```

Parent topic:Linux OS host

Run an example application To run a demo application using J-Link GDB Server application, perform the following steps.

1. Connect the development platform to your PC via USB cable between the DBG USB connector (J1401) and the PC USB connector.
2. Connect 12 V ~ 20 V power supply and J-Link Plus to the device.
3. Switch SW1301[3:0] to the M core boot and ensure that the image is not available on the boot source. For example, 0b1010 for MicroSD boot. Keep the SD slot empty.
4. Open the terminal application on the PC, such as PuTTY or TeraTerm, connect to the debug COM port, see [How to determine COM port](#), and configure the terminal with these settings:
 1. 115200 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity

3. 8 data bits
4. 1 stop bit



5. Open the J-Link GDB Server application. Assuming the J-Link software is installed, the application can be launched from a new terminal for the MIMX9352_M33 device:

```
JLinkGDBServer.exe -jlinkscriptfile Devices\NXP\iMX93\NXP_iMX93_Connect_CortexM33.  
→JLinkScript -device MIMX9352_M33 -if SWD
```

```
-----GDB Server start settings-----  
GDBInit file:          none  
GDB Server Listening port: 2331  
SWO raw output listening port: 2332  
Terminal I/O port:     2333  
Accept remote connection: localhost only  
Generate logfile:      off  
Verify download:       off  
Init regs on start:    off  
Silent mode:           off  
Single run mode:       off  
Target connection timeout: 5000 ms  
-----J-Link related settings-----  
J-Link Host interface:  USB
```

(continues on next page)

(continued from previous page)

```

J-Link script:      Devices\NXP\iMX93\NXP_iMX93_Connect_CortexM33.JLinkScript
J-Link settings file: none
-----Target related settings-----
Target device:      MIMX9352_M33
Target interface:    SWD
Target interface speed: 4000kHz
Target endian:      little

Connecting to J-Link...
J-Link is connected.
Firmware: J-Link V9 compiled May 7 2021 16:26:12
Hardware: V9.60
S/N: 59611220
Feature(s): RDI, GDB, FlashDL, FlashBP, JFlash
Checking target voltage...
Target voltage: 1.98 V
Listening on TCP/IP port 2331
Connecting to target...
Connected to target
Waiting for GDB connection...

```

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

```

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

```

For this example, the path is:

```
*<install\_dir>/boards/mcimx93evk/demo\_apps/hello\_world/armgcc/debug*
```

7. Start the GDB client:

```

$ arm-none-eabi-gdb hello_world.elf
GNU gdb (GNU Tools for Arm Embedded Processors 9-2019-q4-major) 8.3.0.20190709-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello_world.elf...
(gdb)

```

8. Connect to the GDB server and load the binary by running the following commands:

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

```
(gdb) target remote localhost:2331
```

(continues on next page)

(continued from previous page)

```

Remote debugging using localhost:2331
0x00000008 in \_\_isr\_vector \(\)
(gdb) monitor reset
Resetting target
(gdb) monitor halt
(gdb) load
Loading section .interrupts, size 0x240 lma 0x0
Loading section .text, size 0x3ab8 lma 0x240
Loading section .ARM, size size 0x8 lma 0x3cf8
Loading section .init\_array, size 0x4 lma 0x3d00
Loading section .fini\_array, size 0x4 lma 0x3d04
Loading section .data, size 0x64 lma 0x3d08
Start address 0x2f4, load size 15724
Transfer rate: 264 KB/sec, 2620 bytes/write.
\(\gdb\)

```

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

```
(gdb) monitor go
```

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



Note: If the software is already running on the M core, the debugger loading image into TCM may get HardFault or a data verification issue. NXP recommends you to follow the steps above to use the debugger. Repowering the board is required to restart the debugger.

Parent topic:Linux OS host

Parent topic:[Run a demo 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 tool chain Download and run the installer from GNU Arm Embedded Toolchain. This is the actual toolset (in other words, compiler, linker, and so on). The GCC toolchain should correspond to the latest supported version, as described in *MCUXpresso SDK Release Notes*.

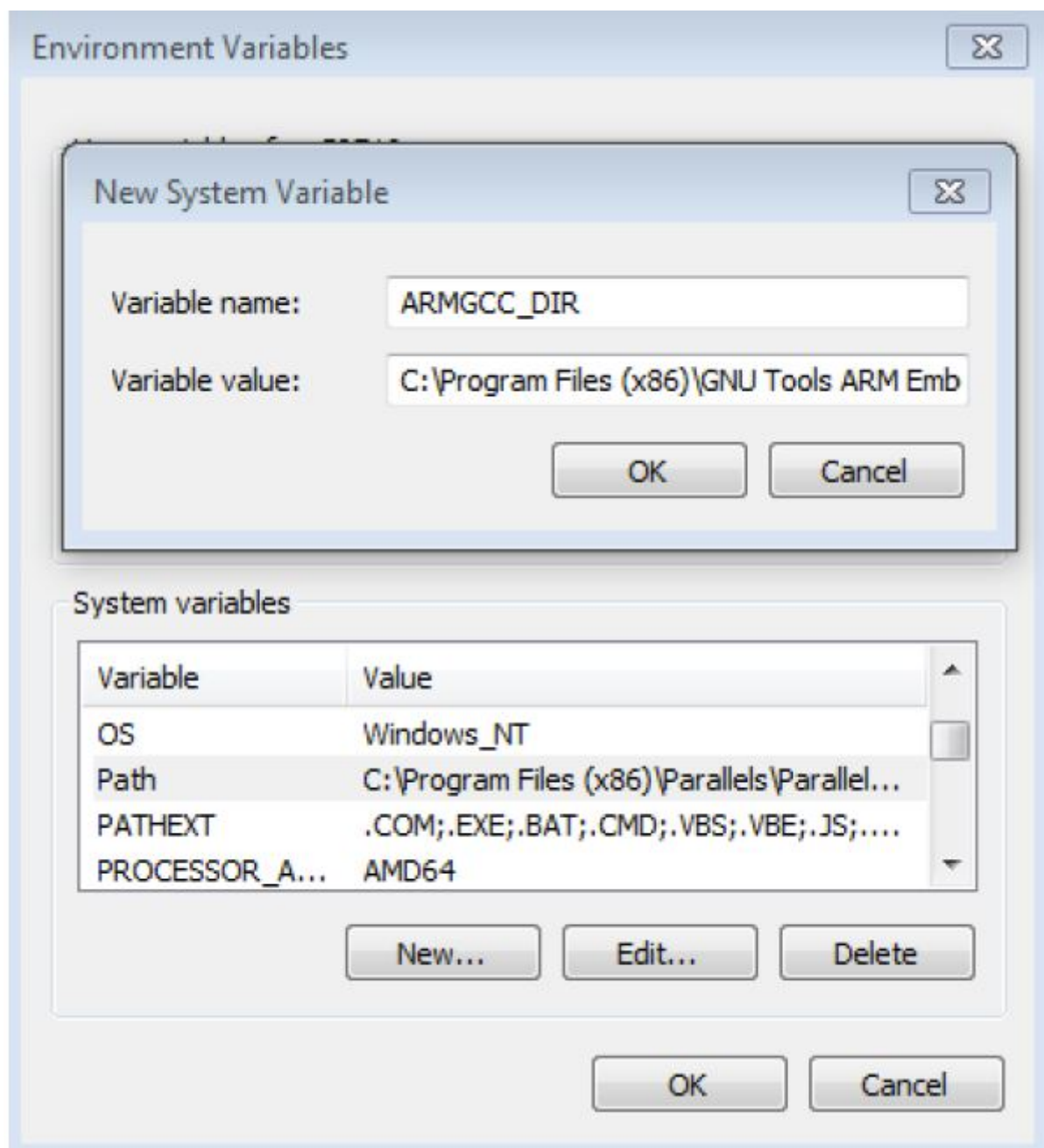
Note: See [Host setup](#) for Windows OS before compiling the application.

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 tool chain installation path.

C:\Program Files (x86)\GNU Tools Arm Embedded\9 2019-q4-major

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

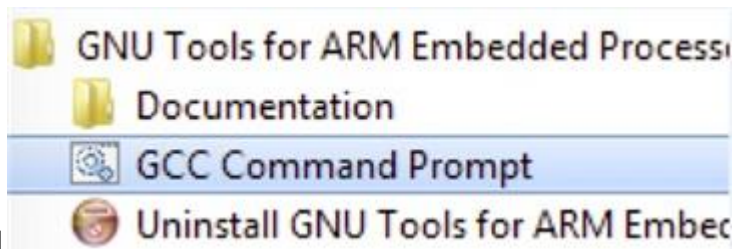


Parent topic:Set up toolchain

Parent topic:Windows OS host

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

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



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

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

For this example, the exact path is:

```
<install_dir>/boards/mcimx93evk/demo_apps/hello_world/armgcc
```

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

```
Building ASM object C:\MakeFiles\hello_world.elf.dir/C:/Users/nxa19635/Downloads/board_MCIMX93-EVK/devices/MIMX9352/utilities/fs1_memcpy.S.obj
[100%] Linking C executable release\hello_world.elf
Memory region  Used Size  Region Size  %age Used
m_interrupts:    1140 B      1144 B      99.65%
m_text:          4608 B     129928 B      3.55%
m_data:          2320 B      116 KB      1.95%
m_m33_suspend_ram: 0 GB         8 KB      0.00%
m_a55_suspend_ram: 0 GB         4 KB      0.00%
[100%] Built target hello_world.elf
C:\Users\nxa19635\Downloads\board_MCIMX93-EVK\boards\mcimx93evk\demo_apps\hello_world\armgcc>IF "" == "" (pause )
Press any key to continue . . .
```

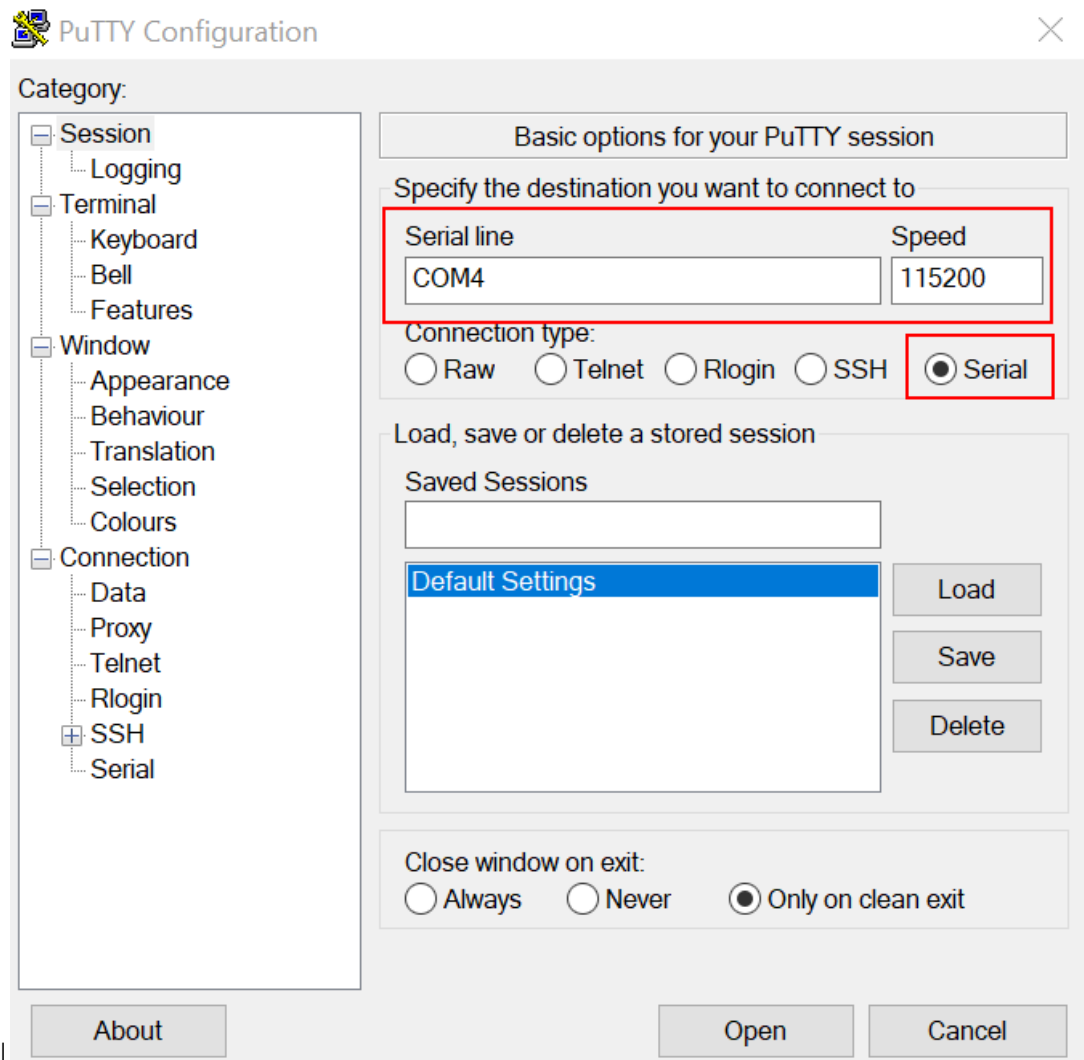
Parent topic:Windows OS host

Run an example application This section describes steps to run a demo application using J-Link GDB Server application.

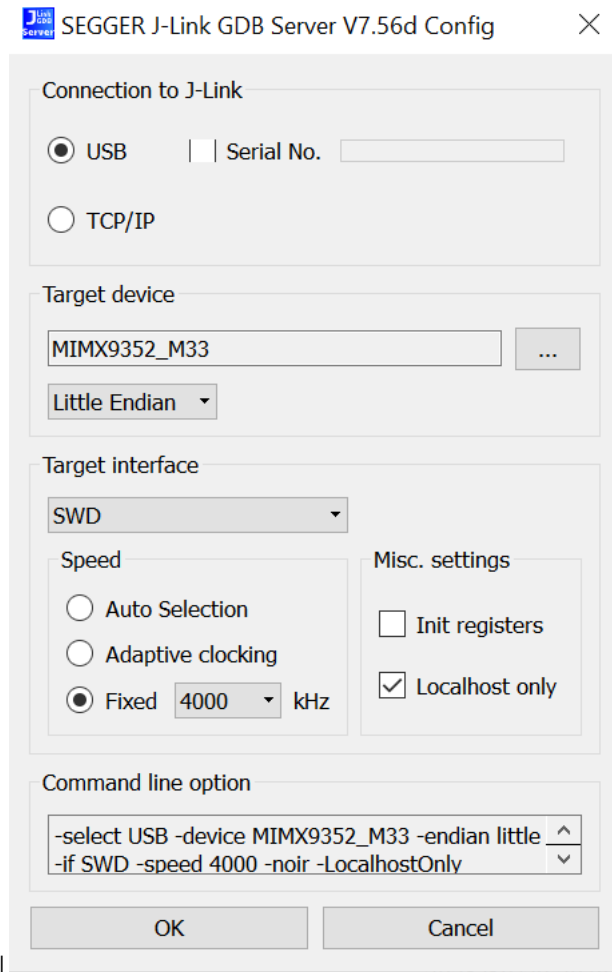
To perform this exercise, the following step must be done.

1. Connect the development platform to your PC via USB cable between the DBG USB connector (J1401) and the PC USB connector.
2. Connect 12 V ~ 20 V power supply and J-Link Plus to the device.
3. Switch SW1301[3:0] to the M core boot and ensure that the image is not available on the boot source. For example, 0b1010 for MicroSD boot. Keep the SD slot empty.
4. Open the terminal application on the PC, such as PuTTY or TeraTerm, connect to the debug COM port, see [How to determine COM port](#), and configure the terminal with these settings:
 1. 115200 baud rate
 2. No parity

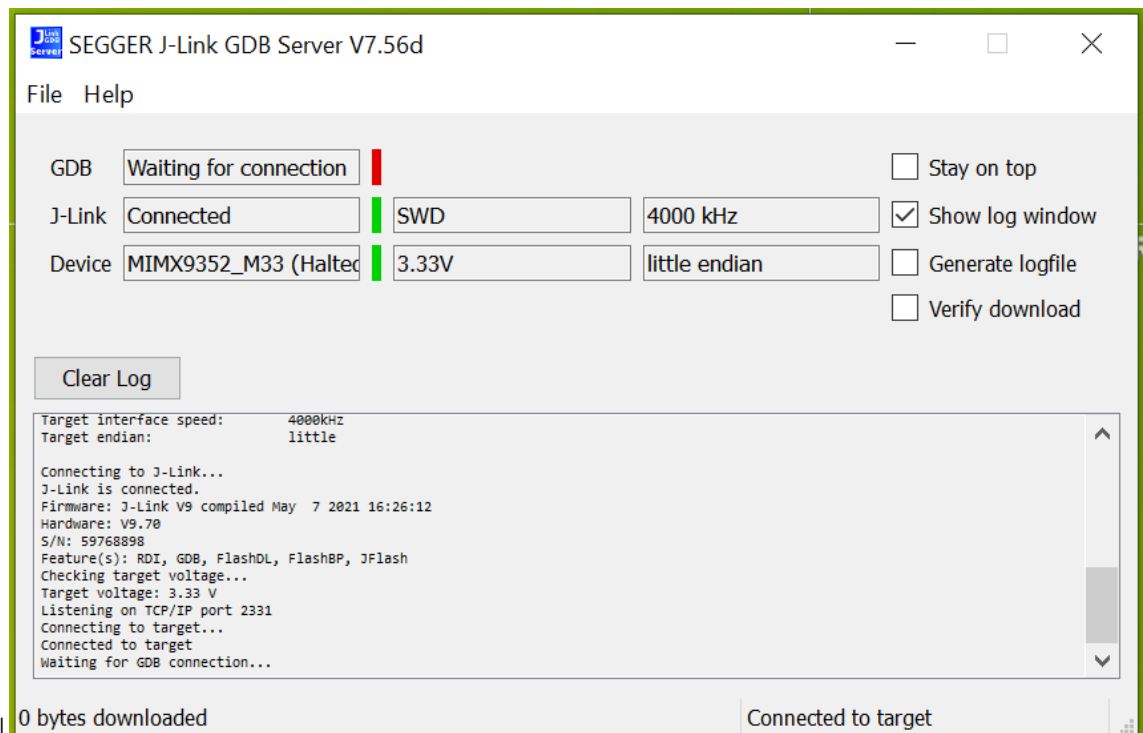
3. 8 data bits
4. 1 stop bit



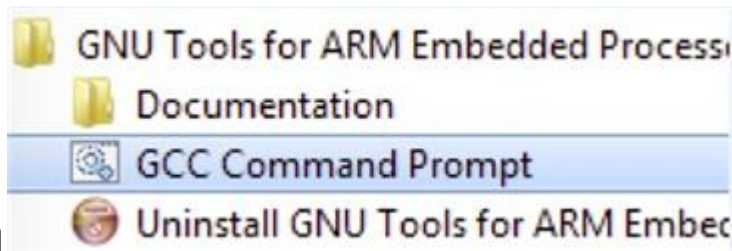
5. Open the J-Link GDB Server application. Assuming the J-Link software is installed, the application can be launched by going to the Windows operating system **Start** menu and selecting **Programs -> SEGGER -> J-Link <version> J-Link GDB Server**.
6. Modify the settings as shown in Figure 2. The target device selection chosen for this example is MIMX9352_M33 .



7. After GDB server is running, the screen should resemble Figure 3:



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



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

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

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

For this example, the path is:

```
<install_dir>/boards/mcimx93evk/demo_apps/hello_world/armgcc/debug
```

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

```

nxa19635@NXL65630 MINGW64 ~/Downloads/board_MCIMX93-EVK/boards/mcimx93evk/demo_apps/hello_world/armgcc/debug
$ arm-none-eabi-gdb.exe hello_world.elf
GNU gdb (GNU Tools for Arm Embedded Processors 9-2019-q4-major) 8.3.0.20190709-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello_world.elf...
(gdb)

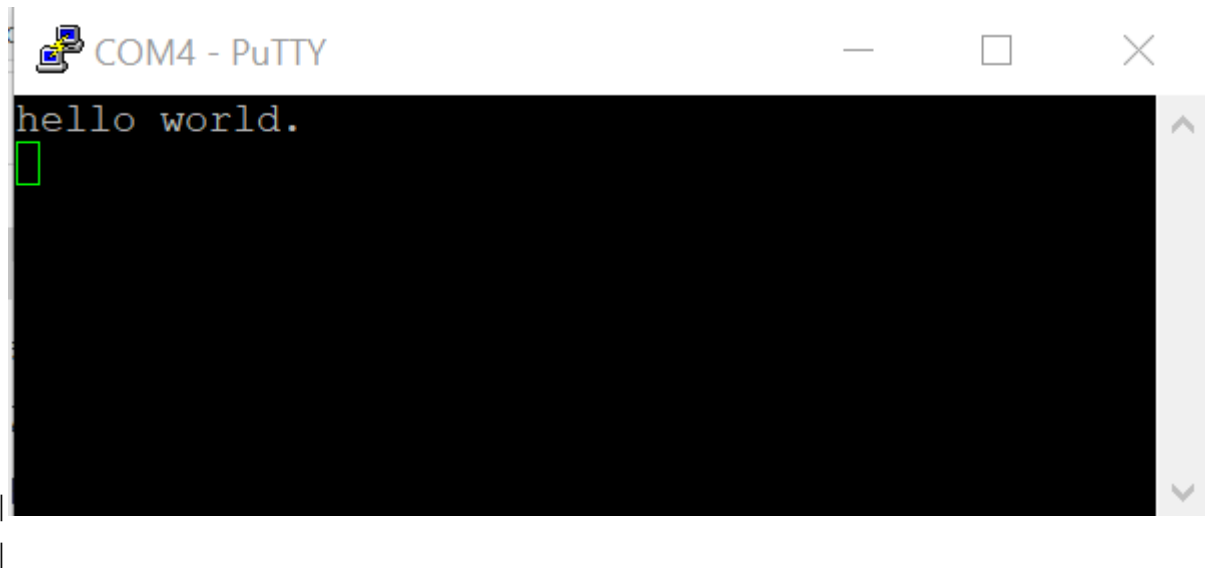
```

11. Run these commands:

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

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

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



Note: If the software is already running on the M core, the debugger loading image into TCM may get HardFault or a data verification issue. NXP recommends you to follow the steps above to use the debugger. Repowering the board is required to restart the debugger.

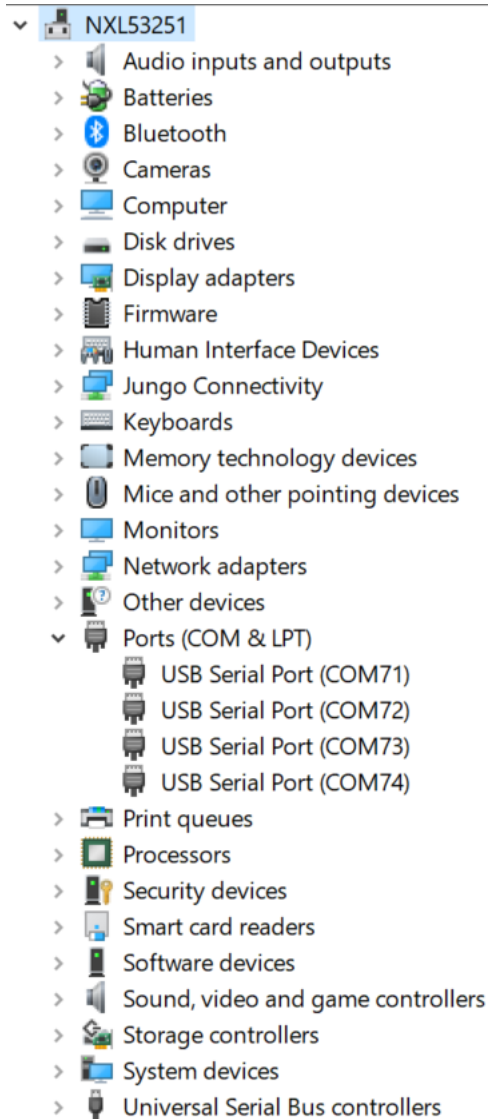
Parent topic: Windows OS host

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

Running an application by U-Boot

This section describes the steps to write a bootable SDK bin file to TCM with the prebuilt U-Boot image for the i.MX processor. The following steps describe how to use the U-Boot:

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 & LPT)**. Assume that the ports are COM71 - COM74. COM73 is for the debug message from the Cortex-A55 and COM74 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. Build the application (for example, `hello_world`) to get the bin file (`hello_world.bin`).
4. Prepare an SD card with the prebuilt U-Boot image and copy bin file (`hello_world.bin`) into the SD card. Then, insert the SD card to the target board. Make sure to use the default boot SD slot and check the dipswitch configuration.
5. Open your preferred serial terminals for the serial devices, setting the speed to 115200 bps, 8 data bits, 1 stop bit (115200, 8N1), no parity, then power on the board.
6. Power on the board and hit any key to stop autoboot in the A55 terminal.
7. Enter to U-Boot command line mode. You can then write the image and run it from TCM with the following commands:
 - `fatload mmc 1:1 80000000 hello_world.bin; cp.b 0x80000000 0x201e0000 0x10000;`
 - `bootaux 0x201e0000 0`


```

U-Boot 2022.04-imx_v2022.04+g3d51d4f18b (Jun 30 2022 - 06:55:33 +0000)

CPU:   i.MX93 rev1.0
Model: NXP i.MX93 11X11 EVK board
DRAM:  2 GiB
TCPC:  Vendor ID [0x1fc9], Product ID [0x5110], Addr [I2C0 0x52]
SNK.Power3.0 on CC1
PDO 0: type 0, 5000 mV, 3000 mA [E]
PDO 1: type 0, 9000 mV, 3000 mA []
PDO 2: type 0, 15000 mV, 3000 mA []
PDO 3: type 0, 20000 mV, 2250 mA []
Requesting PDO 3: 20000 mV, 750 mA
Source accept request
PD source ready!
tcpc_pd_receive_message: Polling ALERT register, TCPC_ALERT_RX_STATUS bit failed
, ret = -62
TCPC:  Vendor ID [0x1fc9], Product ID [0x5110], Addr [I2C0 0x51]
TCPC:  Vendor ID [0x1fc9], Product ID [0x5110], Addr [I2C0 0x50]
Core:  204 devices, 26 uclasses, devicetree: separate
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Loading Environment from MMC... *** Warning - bad CRC, using default environment

[*-]Video Link 0adv7535_mipi2hdmi hdmi@3d: Can't find cec device id=0x3c
fail to probe panel device hdmi@3d
fail to get display timings
probe video device failed, ret -19

    [0] lcd-controller@4ae30000, video
    [1] dsi@4ae10000, video_bridge
    [2] hdmi@3d, panel
adv7535_mipi2hdmi hdmi@3d: Can't find cec device id=0x3c
fail to probe panel device hdmi@3d
fail to get display timings
probe video device failed, ret -19
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc1 is current device
UID: 0x9d96f518 0x944d9627 0xe7f8e89 0xc47fcb39
flash target is MMC:1
Net:
Warning: ethernet@428a0000 (eth1) using random MAC address - 82:dc:44:72:38:9d

Warning: ethernet@42890000 (eth0) using random MAC address - ce:ff:2c:ad:e5:e6
eth0: ethernet@42890000 [PRIME], eth1: ethernet@428a0000
Fastboot: Normal
Normal Boot
Hit any key to stop autoboot:  0
=>
=> fatload mmc 0:1 80000000 m33_image.bin
8788 bytes read in 6 ms (1.4 MiB/s)
=> cp.b 0x80000000 0x201e0000 0x10000
=> bootaux 0x201e0000 0
## Starting auxiliary core stack = 0x20020000, pc = 0x0FFE1B09...
=> █

```

8. The `hello_world` application is now running and a banner is displayed on the M33 terminal. If this is not true, check your terminal settings and connections.



Program flash.bin to SD/eMMC with UUU

This section describes the steps to use the UUU to run the example applications provided in the MCUXpresso SDK. Download the flash.bin to emmc/sd with UUU. The hello_world demo application targeted for the i.MX 93 hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

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

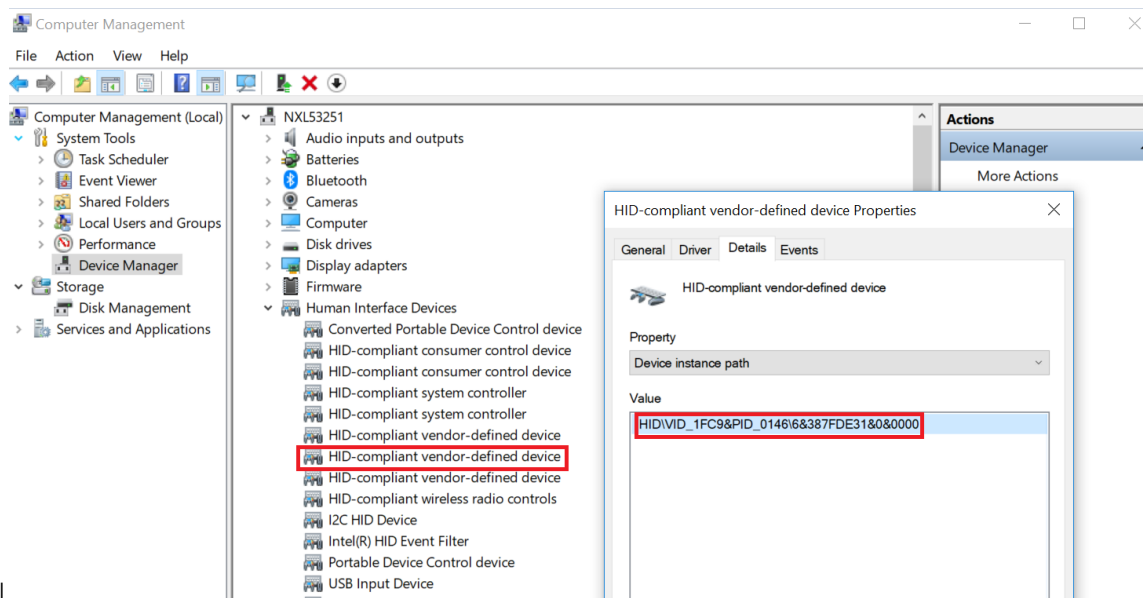
Download the Universal Upgrade Utility The Universal Upgrade Utility (UUU) is an upgraded version of MfgTool. It is a command line tool that aims at installing the bootloader to various storage including SD, QSPI, and so on, for i.MX series devices with ease.

The tool can be accessed from corresponding Linux BSP release. Download uuu.exe for Windows OS, or download UUU for Linux. Configure the path so that the executable can later be called anywhere in the command line.

Parent topic:Set up environment

Switch to Download Mode The board needs to be in Download Mode mode for UUU to download images:

1. Set the board boot mode to Download Mode [SW1301[3:0] = 0011(Single Boot Mode)/1011(Low Power Boot Mode)].
2. Connect the development platform to your PC via USB cable between the DBG USB connector (J1401) and the PC USB connector.
3. Connect J403 (USB1) to PC USB connector for downloading.
4. The PC recognizes the i.MX 93 device as (VID:PID)=(1FC9:0146), as shown in Figure 1.



Parent topic:Set up environment

Parent topic:[Program flash.bin to SD/eMMC with UUU](#)

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

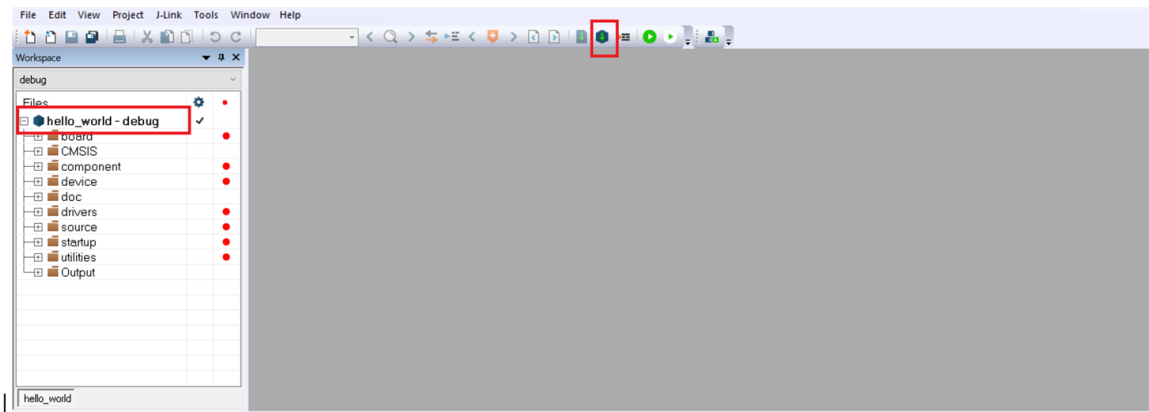
1. If not already done, open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

Using the i.MX 93 EVK board as an example, the workspace is located in:

```
<install_dir>/boards/mcimx93evk/demo_apps/hello_world/iar/hello_world.eww
```

2. Select the desired build target from the drop-down. For this example, select **hello_world – debug**.
3. To build the demo application, click **Make**.

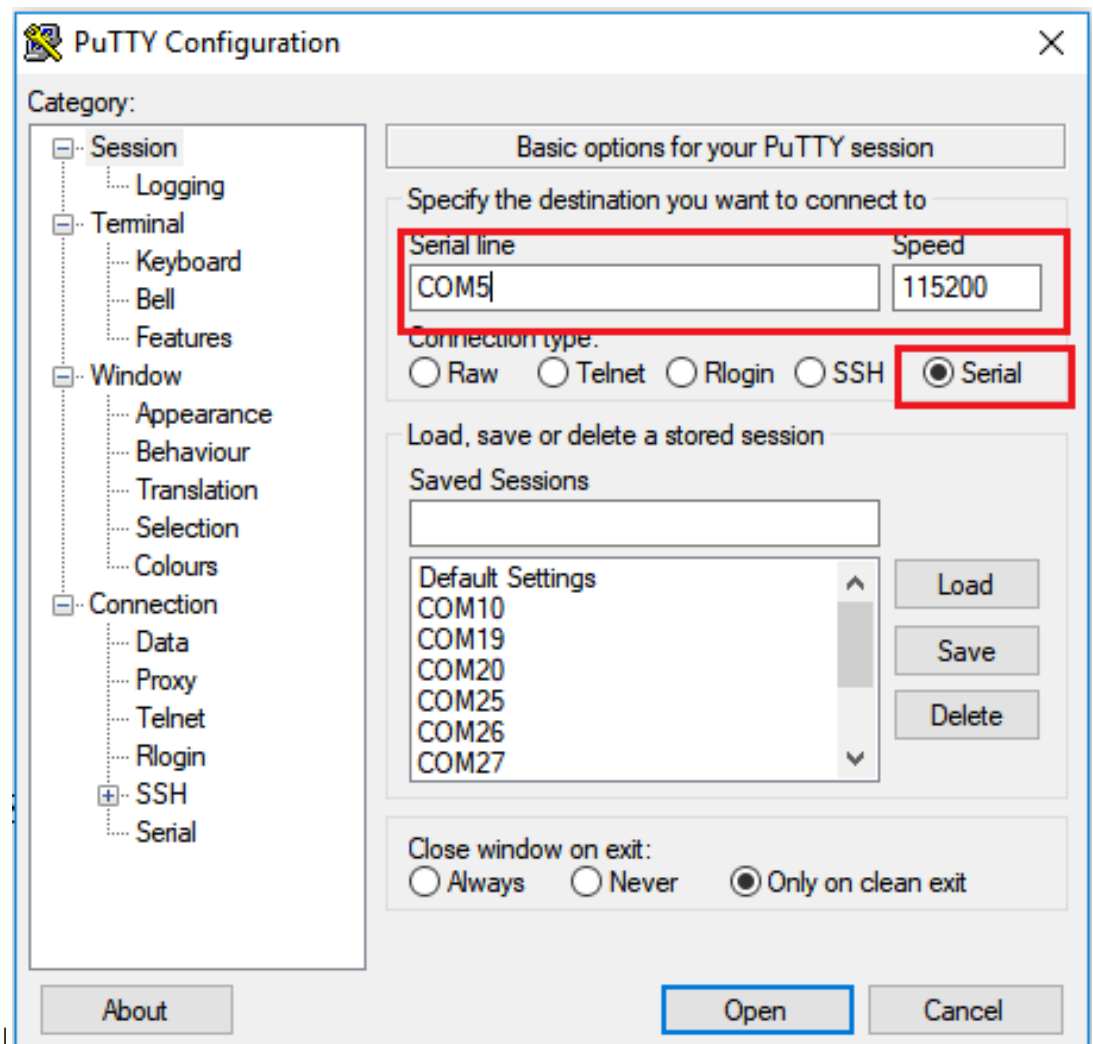


4. The build completes without errors.

Parent topic: [Program flash.bin to SD/eMMC with UUU](#)

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

1. Connect the development platform to your PC via USB cable between the DBG USB connector (J1401) and the PC. It provides console output while using UUU.
2. Connect the J403 (USB1) connector and the PC. It provides the data path for UUU.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 baud rate
 2. No parity
 3. 8 data bits



4. 1 stop bit |

4. Get the boot images and the imx-mkimage source repository from corresponding Linux BSP release. The boot images required to be put into imx-mkimage/i.MX9 are:

- u-boot-imx93evk.bin-sd (rename to u-boot.bin)
- u-boot-spl.bin-imx93evk-sd (rename to u-boot-spl.bin)
- bl31-imx93.bin (rename to bl31.bin)
- mx93a0-ahab-container.img
- lpddr4_dmem_1d_v202201.bin
- lpddr4_dmem_2d_v202201.bin
- lpddr4_imem_1d_v202201.bin
- lpddr4_imem_2d_v202201.bin

5. Make flash.bin with imx-mkimage.

make SOC=iMX9 flash_singleboot_m33 (for single boot mode)

or

make SOC=iMX9 flash_lpboot (for low power boot mode)

6. Type the UUU command to the flash image.

uuu -b emmc flash.bin (for single boot on eMMC)

`uuu -b sd flash.bin` (for single boot on SD)

For low power boot, a single boot flash.bin is needed besides the target flash.bin.

`uuu -b emmc <singleboot flash.bin> flash.bin` (for lowpower boot on eMMC)

`uuu -b sd <singleboot flash.bin> flash.bin` (for lowpower boot on SD)

The UUU puts the platform into fast boot mode and automatically flashes the target boot-loader to emmc/sd. The command line and fast boot console is as shown in Figure 2.

```

Switch to partitions #0, OK
mmc0(part 0) is current device
flash target is MMC:0
Net:
Warning: ethernet@428a0000 MAC addresses don't match:
Address in ROM is      01:02:03:04:05:06
Address in environment is 36:e5:5c:42:fa:e5

Warning: ethernet@42890000 MAC addresses don't match:
Address in ROM is      01:02:03:04:05:06
Address in environment is be:8a:6c:54:c1:5a
eth0: ethernet@42890000 [PRIME], eth1: ethernet@428a0000
Fastboot: Normal
Normal Boot
Hit any key to stop autoboot: 0
=> fastboot 0
Failed to configure default pinctrl
switch to partitions #0, OK
mmc0(part 0) is current device
Starting download of 1014784 bytes
.....
downloading of 1014784 bytes finished
writing to partition 'bootloader'
Initializing 'bootloader'
switch to partitions #1, OK
mmc0(part 1) is current device
Writing 'bootloader'

MMC write: dev # 0, block # 0, count 1982 ... 1982 blocks written: OK
Writing 'bootloader' DONE!

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB4

sudo uuu -b emmc flash.bin
uuu (Universal Update Utility) for nxp imx cnips -- libuuu_1.4.224-0-g70d1e85

Success 1   Failure 0

1:141    7/ 7 [Done]                ] FB: Done

nxf49783@biwen:/mnt/home/nxf49783/logs/imx/imx93/imx93evk/verify_rpmsg$

```

7. Then, power off the board and change the boot mode to the corresponding one.

- For single-boot mode:
 - when boot device is emmc, then $\text{SW1301}[3:0] = 0000$;
 - when boot device is sd, then $\text{SW1301}[3:0] = 0010$.
- For low-power boot mode:
 - when boot device is emmc, then $\text{SW1301}[3:0] = 1000$;
 - when boot device is sd, then $\text{SW1301}[3:0] = 1010$.

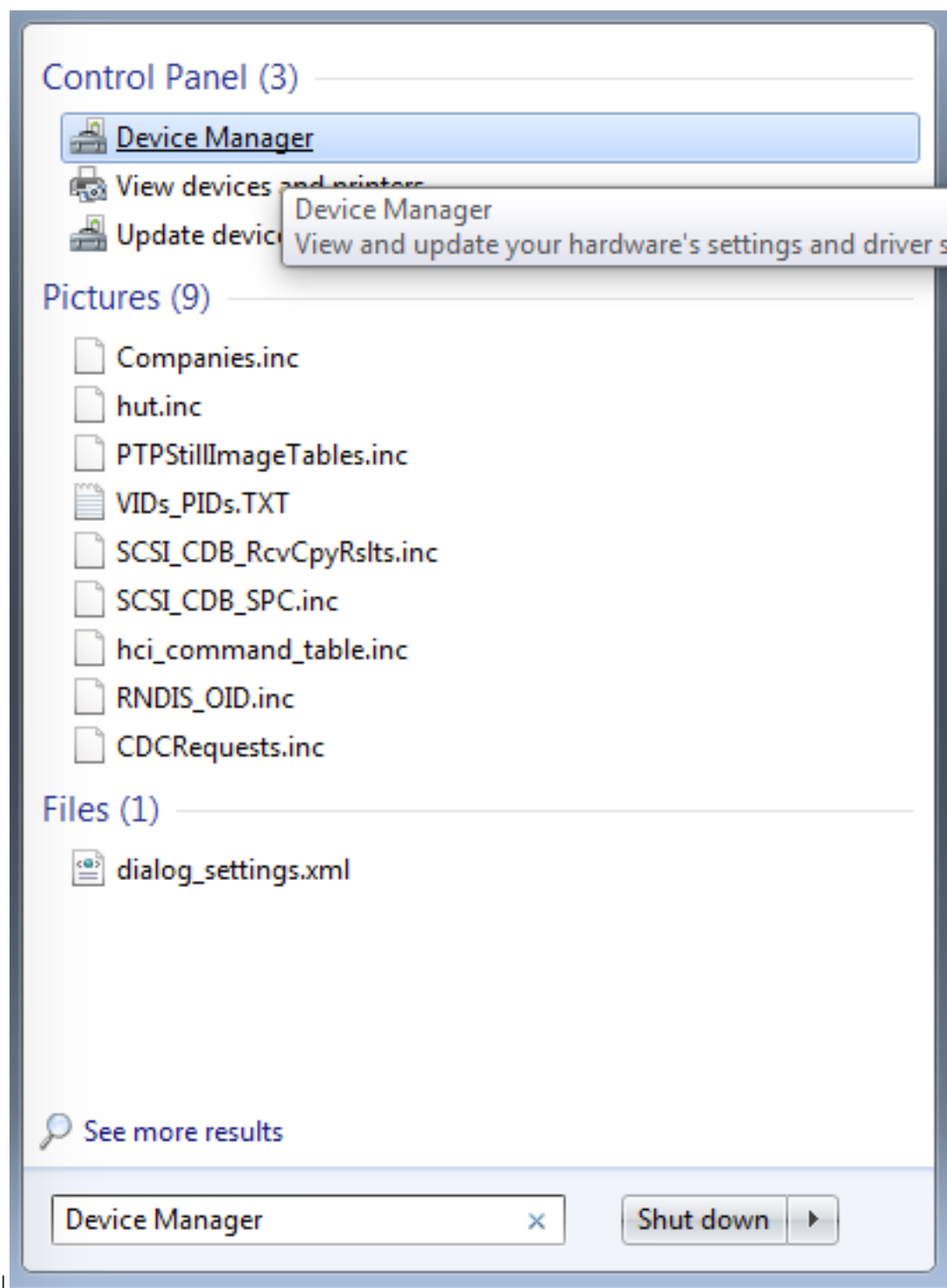
8. Power on the board again.

Parent topic: [Program flash.bin to SD/eMMC with UUU](#)

How to determine COM port

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

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



2. In the **Device Manager**, expand the **Ports (COM & LPT)** section to view the available ports. Depending on the NXP board you're using, the COM port can be named differently.
 1. **USB-UART** interface



Host setup

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:

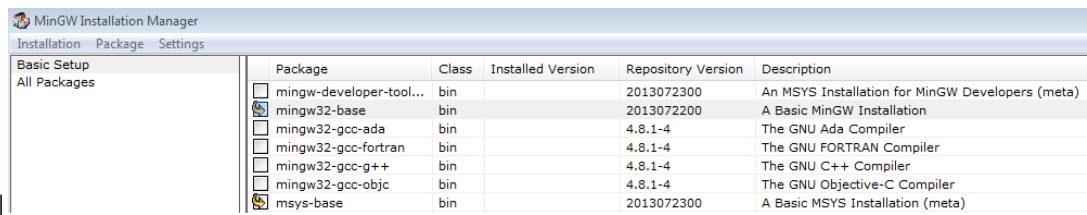
- MinGW

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 cannot 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](host_setup.md #ADDINGPATH). If the path is not set correctly, the toolchain does not work.

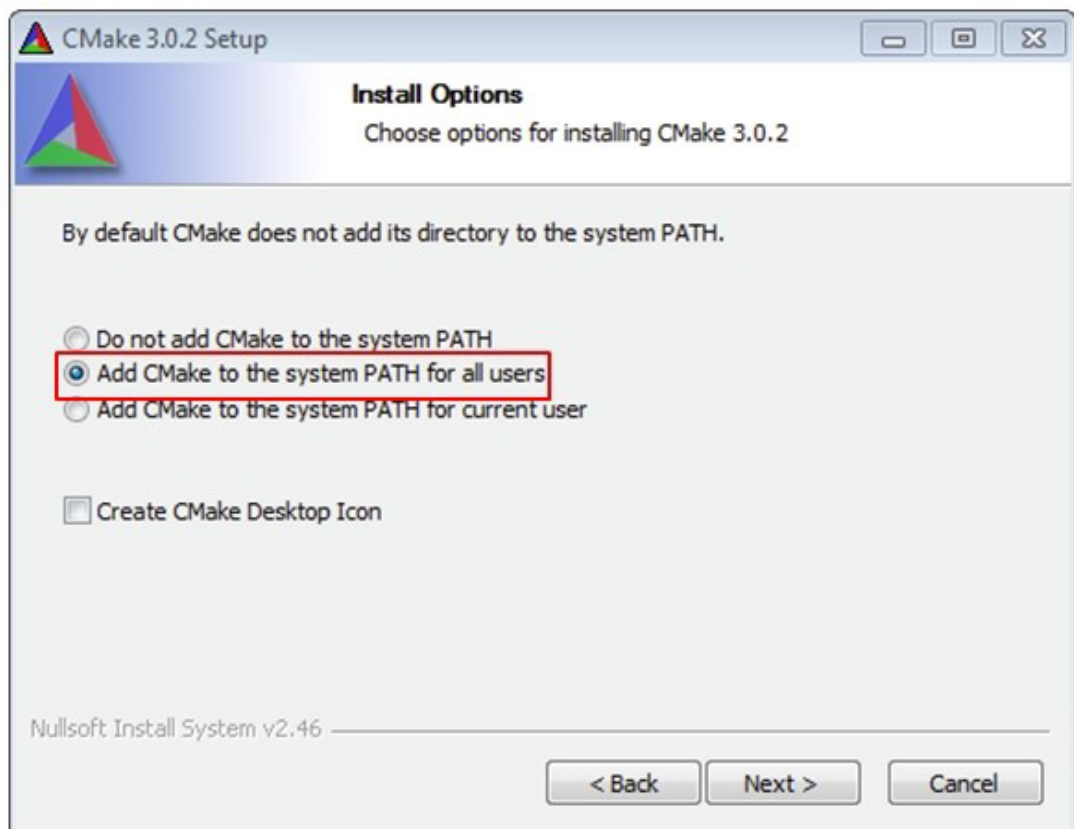
(continues on next page)

(continued from previous page)

****Note:**** If you have `C:\MinGW\msys\x.x\bin` in your PATH variable \ (as required by KSDK 1.0.0\),
 → 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. Install CMake, ensuring that the option **Add CMake to system PATH** is selected when installing. The user chooses to select whether it is installed into the PATH for all users or just the current user. In this example, it is installed for all users.



3. Follow the remaining instructions of the installer.
4. You may need to reboot your system for the PATH changes to take effect.

1.3 Getting Started with MCUXpresso SDK GitHub

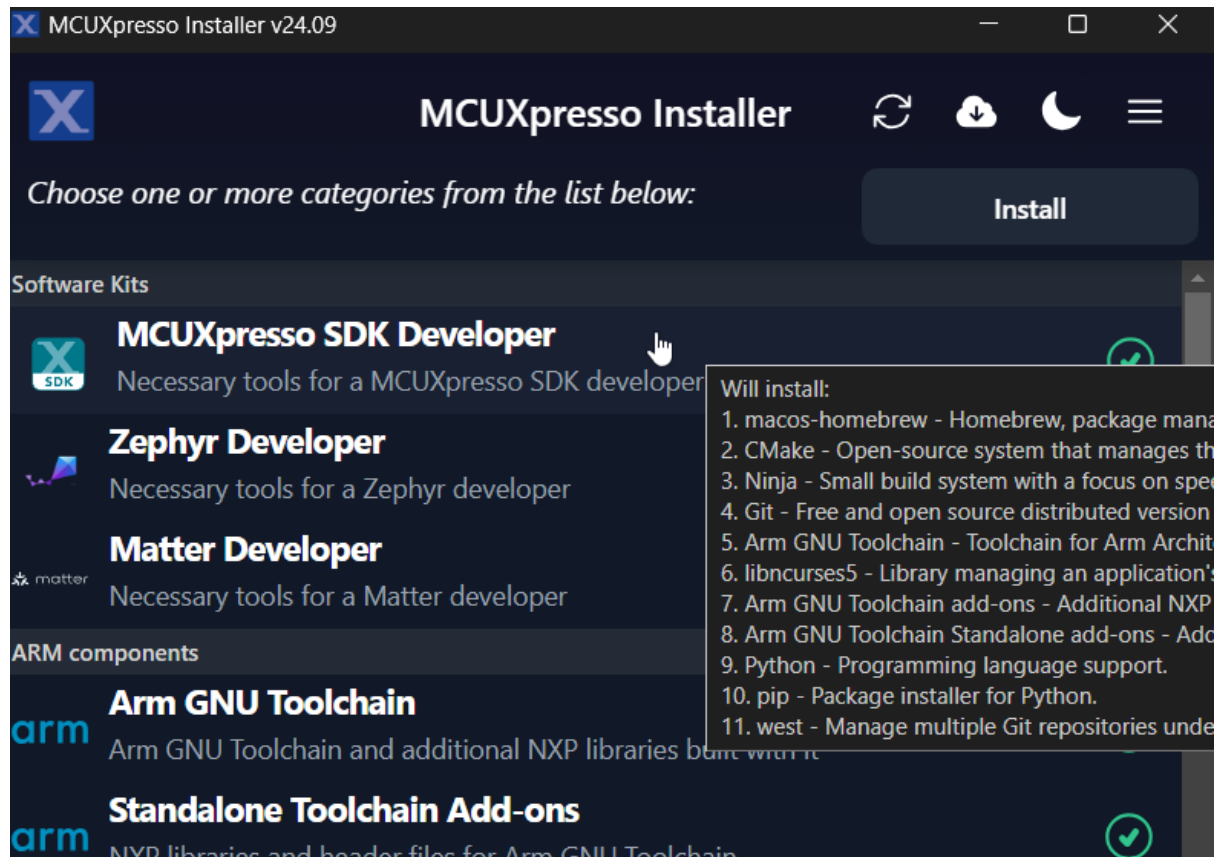
1.3.1 Getting Started with MCUXpresso SDK Repository

Installation

NOTE

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

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



Alternative: Manual Installation

Basic tools

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

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

Then configure your username and email using the commands:

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

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

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

West Please use the west version equal or greater than 1.2.0

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

Build And Configuration System

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-  arm
-  common
-  install-info

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

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

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

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

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

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

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

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

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

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

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

- macOS:

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

Get MCUXpresso SDK Repo

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

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

# Update the west projects
cd mcuxpresso-sdk
west update

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

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

1. Navigate to the core directory:

```
cd mcuxsdk
```

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

```
python -m venv .venv

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

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

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

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

3. Install the required Python packages:

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

Explore Contents

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

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

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

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

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

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

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

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

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

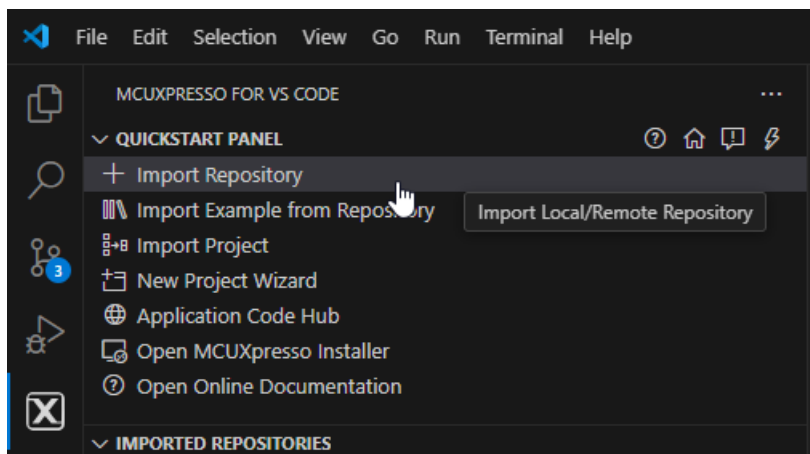
Run a demo using MCUXpresso for VS Code

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

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

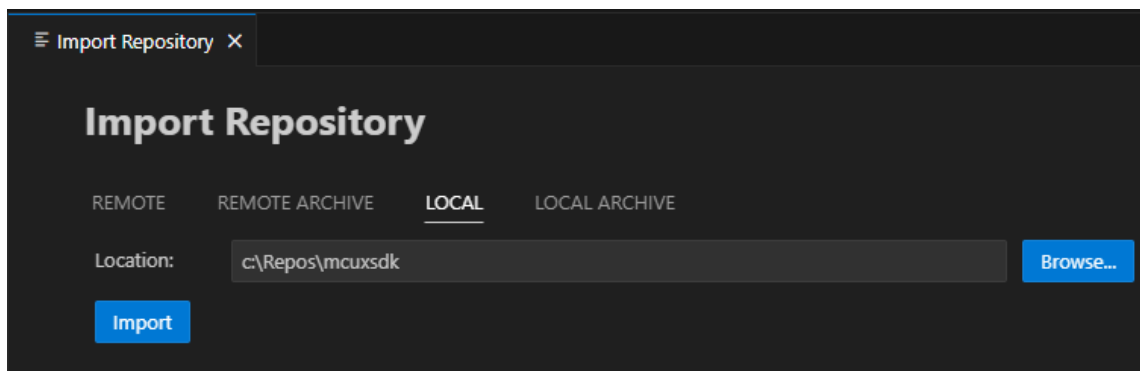
To build an example application:

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

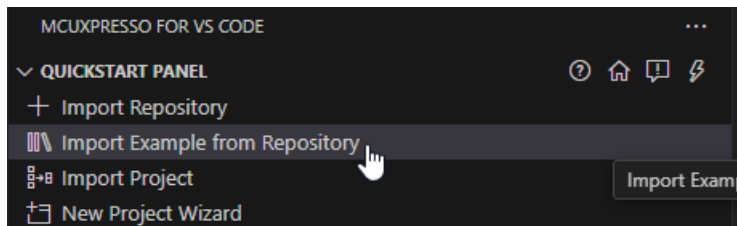


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

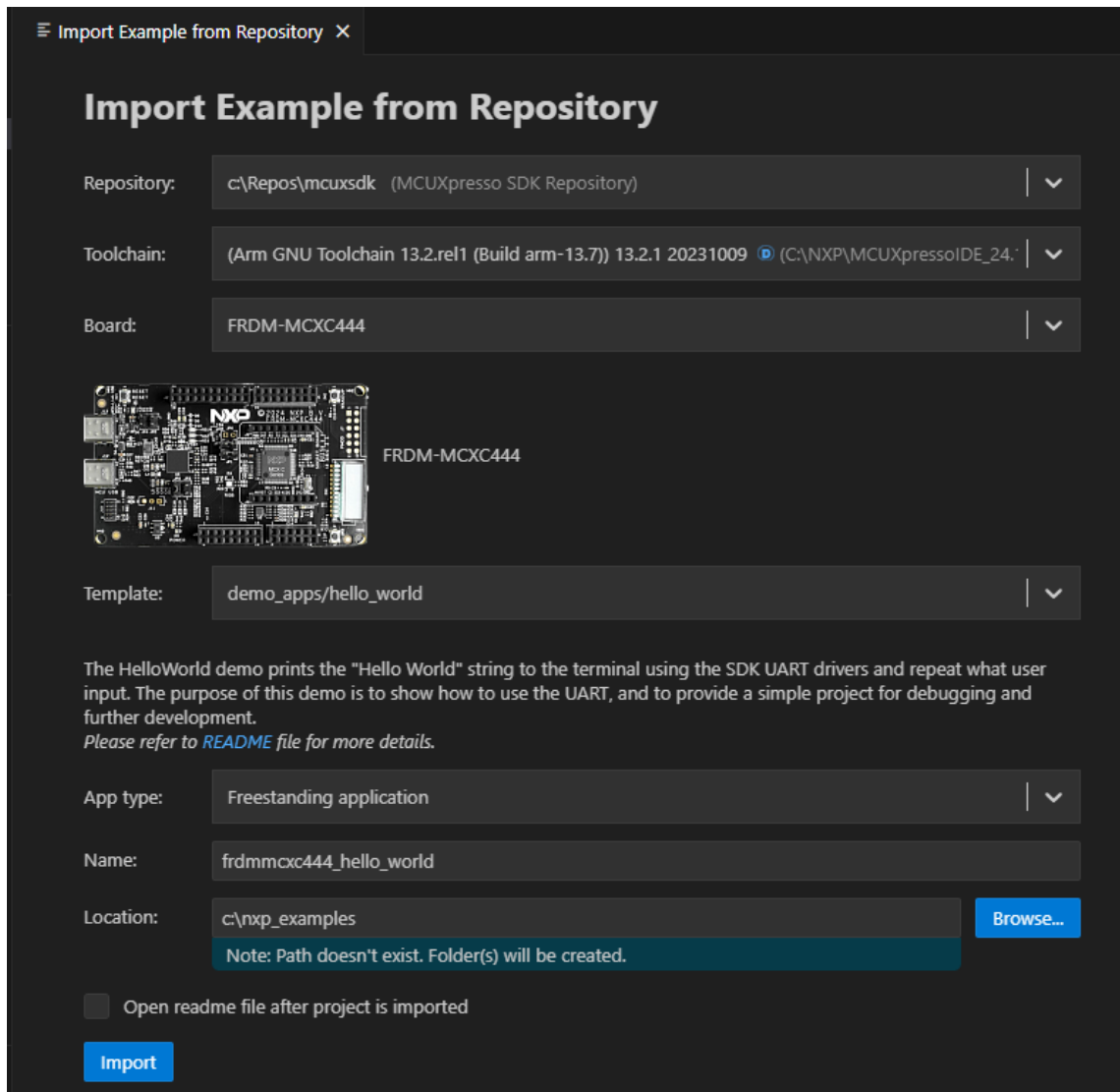
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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

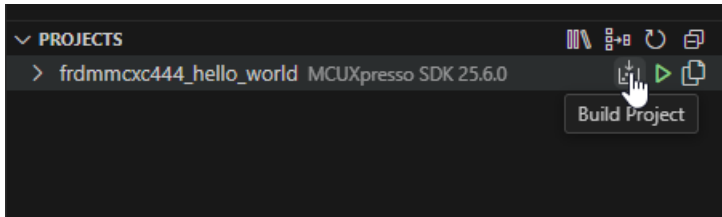


In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.

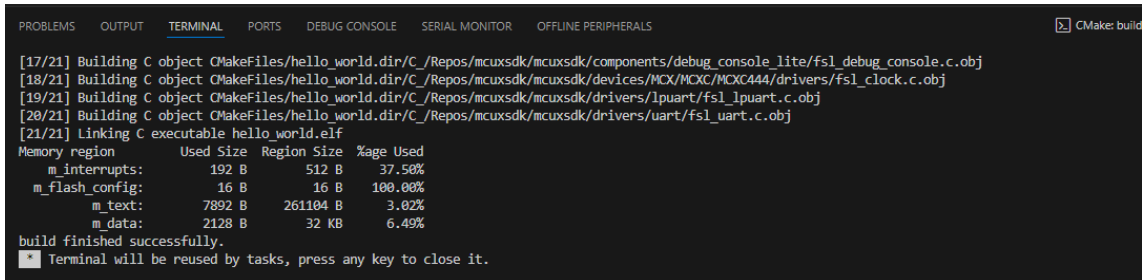


Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Freestanding applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Freestanding examples can be imported to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

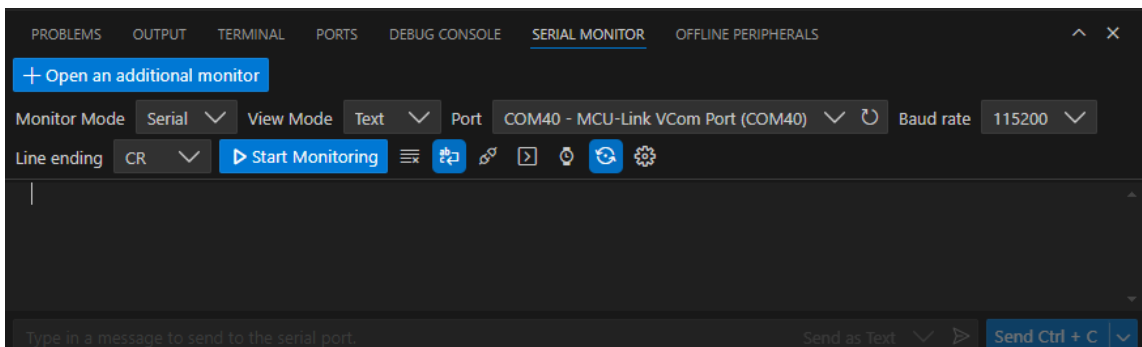


The integrated terminal will open at the bottom and will display the build output.

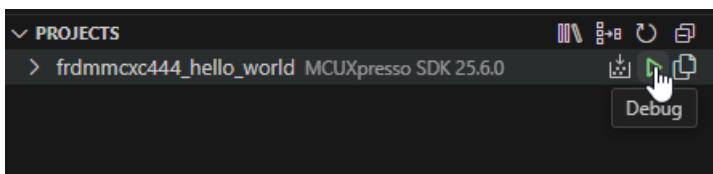


Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

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.

```

C hello_world.c x
frdm-mxc444_hello_world > examples > demo_apps > hello_world > C hello_v
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

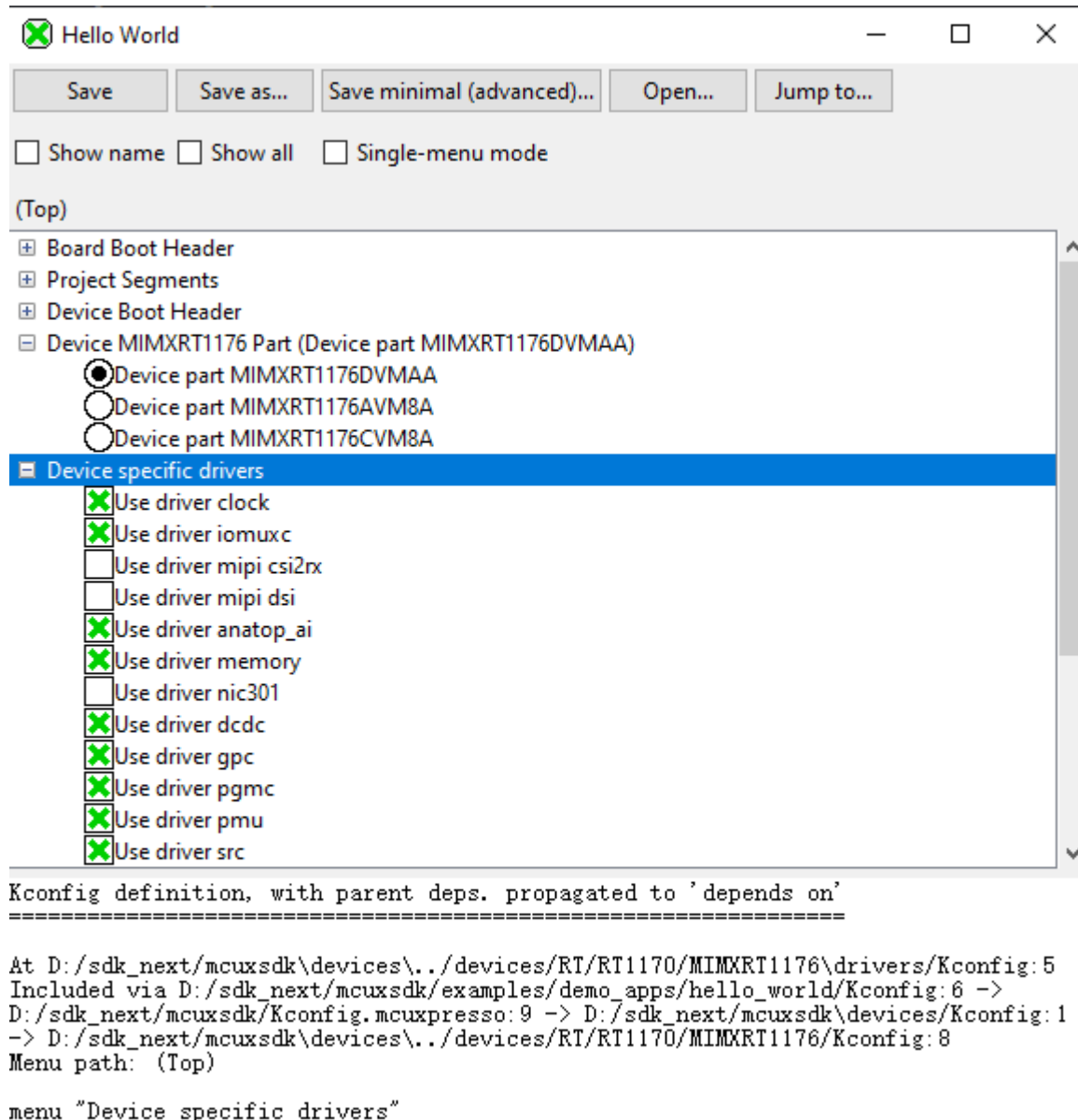
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

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:

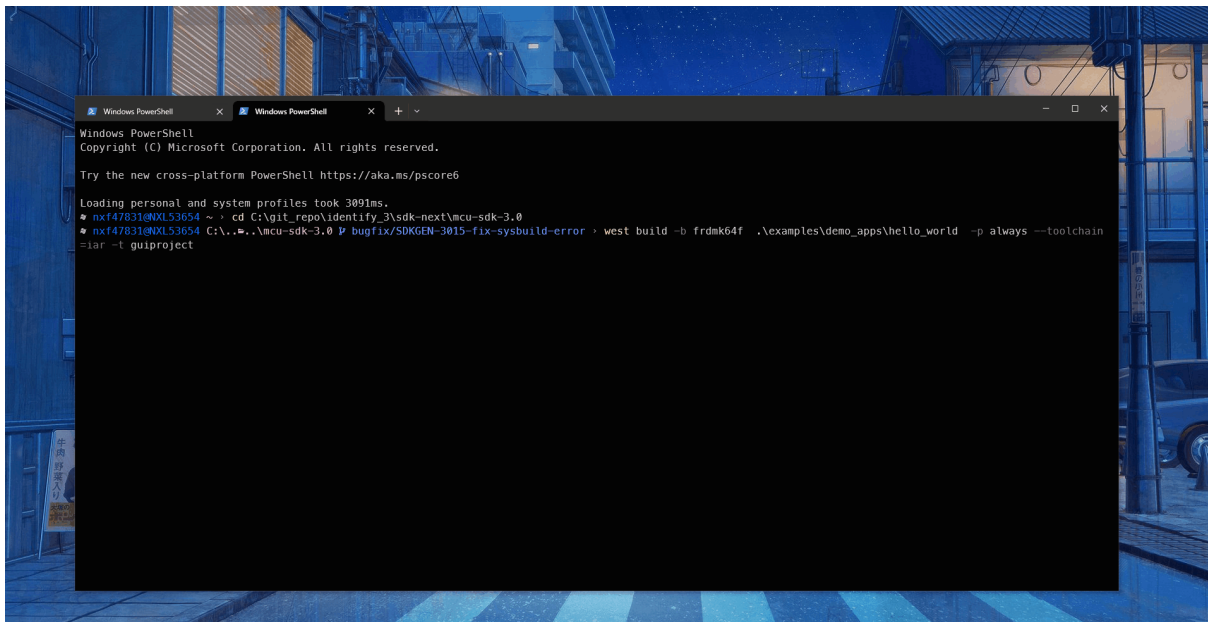
```
west debug -r linkserver
```

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 v25.09
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

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

De- vel- op- ment board	MCU devices			
MCIM EVK	MIMX9331AVTXM, MIMX9332AVTXM, MIMX9351AVTXM, MIMX9352AVTXM, MIMX9301CVVXD,	MIMX9331CVVXM, MIMX9332CVVXM, MIMX9351CVVXM, MIMX9352CVVXM, MIMX9301DVVXD,	MIMX9331DVVXM, MIMX9332DVVXM, MIMX9351DVVXM, MIMX9352DVVXM , MIMX9302CVVXD,	MIMX9331XVVXM, MIMX9332XVVXM, MIMX9351XVVXM, MIMX9352XVVXM, MIMX9302DVVXD

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.

IEC60730B Safety Library NXP IEC60730B Safety Library

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.

Voice Seeker (no AEC) VoiceSeeker is a multi-microphone voice control audio front-end signal processing solution. VoiceSeeker is not featuring acoustic echo cancellation (AEC).

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

eIQ The package contains several example applications using the eIQ TensorFlow Lite for Microcontrollers library.

eIQ machine learning SDK containing:

- Arm CMSIS-NN library (neural network kernels optimized for Cortex-M cores)
- Inference engines:
 - TensorFlow Lite Micro
 - DeepView RT
- Example code for TensorFlow Lite Micro, Glow, and DeepView RT

lwIP The lwIP TCP/IP stack is pre-integrated with MCUXpresso SDK and runs on top of the MCUXpresso SDK Ethernet driver with Ethernet-capable devices/boards.

For details, see the *lwIP TCP/IP Stack and MCUXpresso SDK Integration User's Guide* (document MCUXSDKLWIPUG).

lwIP is a small independent implementation of the TCP/IP protocol suite.

llhttp HTTP parser llhttp

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

Release contents

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

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

Known Issues

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

SEGGER J-Link debugger usage problem

When an M core software is already running, it is possible to get HardFault or data verification issue during loading image into TCM by debugger.

The following steps are recommended to use the J-Link debugger.

1. Configure switch SW1301 to M core boot; low-power boot. Ensure that there is no image on the boot source.
2. Power the board and start the debugger for use.
3. To restart the debugger, stop the debugger, power off the board, and repeat step 2.

eDMA examples accessing AIPS peripheral bridge memory must run through U-Boot loading method

Non-secure access to Arm IP Bus (AIPS) must be configured in Trusted Resource Domain Control (TRDC) for enhanced direct memory access (eDMA) controller. Due to the limitation that Sentinel ROM can release TRDC only once, such examples must run through U-Boot loading method after Trusted Firmware-A (TF-A) configuring TRDC.

To make low-power boot mode work for only M core in such example, you need to implement the request of the TRDC release and configure TRDC. However, it will break the single boot mode with TF-A/Linux BSP.

The following eDMA examples need access to AIPS.

- cmsis_lpi2c_edma_b2b_transfer_master
- cmsis_lpi2c_edma_b2b_transfer_slave

- cmsis_lpuart_edma_transfer
- flexcan_loopback_edma_transfer
- lpi2c_edma_b2b_transfer_master
- lpi2c_edma_b2b_transfer_slave
- lpuart_edma_transfer
- pdm_edma_transfer
- sai_edma_transfer

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

CACHE XCACHE

[2.0.4]

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

[2.0.3]

- Bug Fixes
 - Fixed CERT INT30-C violations.

[2.0.2]

- Bug Fixes
 - Updated `XCACHE_InvalidateCacheByRange()`, `XCACHE_CleanCacheByRange()`, `XCACHE_CleanInvalidateCacheByRange()` in case of startAddr equal to endAddr.

[2.0.1]

- Improvements
 - Check input parameter “size_byte” must be larger than 0.

[2.0.0]

- Initial version.
-

COMMON

[2.6.1]

- Improvements
 - Support Cortex M23.

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.3]

- New Features
 - Added NETC into status group.

[2.3.2]

- Improvements
 - Make driver aarch64 compatible

[2.3.1]

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

[2.3.0]

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

[2.2.10]

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

[2.2.9]

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

[2.2.8]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

EDMA**[2.10.7]**

- Improvements
 - Add condition to fix build warnings(array subscript 4 is above array bounds of 'edma_handle_t *[4][64]')
- Bug Fixes
 - Fixed the EDMA header index retrieval error caused by done bit calculation mistake issue.

[2.10.6]

- Improvements
 - Add macro FSL_FEATURE_EDMA_HAS_EDMA_TCD_CLOCK_ENABLE to enable tcd clocks in EDMA_Init function.

[2.10.5]

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

[2.10.4]

- Improvements
 - Add new MP register macros to ensure compatibility with different devices.
 - Add macro DMA_CHANNEL_ARRAY_STEPn to adapt to complex addressing of edma tcd registers.

[2.10.3]

- Bug Fixes
 - Clear interrupt status flags in EDMA_CreateHandle to avoid triggering interrupt by mistake.

[2.10.2]

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

[2.10.1]

- Bug Fixes
 - Fixed EDMA_GetRemainingMajorLoopCount may return wrong value issue.
 - Fixed violations of the MISRA C-2012 rules 13.5, 10.4.

[2.10.0]

- Improvements
 - Modify the structures `edma_core_mp_t`, `edma_core_channel_t`, `edma_core_tcd_t` to adapt to `edma5`.
 - Add TCD register macro to facilitate confirmation of `tcd` type.
 - Modify the mask macro to a fixed value.
 - Add `EDMA_TCD_TYPE` macro to determine `edma tcd` type.
 - Add extension API to the following API to determine `edma tcd` type.
 - * `EDMA_ConfigChannelSoftwareTCD` -> `EDMA_ConfigChannelSoftwareTCDExt`
 - * `EDMA_TcdReset` -> `EDMA_TcdResetExt`
 - * `EDMA_TcdSetTransferConfig` -> `EDMA_TcdSetTransferConfigExt`
 - * `EDMA_TcdSetMinorOffsetConfig` -> `EDMA_TcdSetMinorOffsetConfigExt`
 - * `EDMA_TcdSetChannelLink` -> `EDMA_TcdSetChannelLinkExt`
 - * `EDMA_TcdSetBandWidth` -> `EDMA_TcdSetBandWidthExt`
 - * `EDMA_TcdSetModulo` -> `EDMA_TcdSetModuloExt`
 - * `EDMA_TcdEnableAutoStopRequest` -> `EDMA_TcdEnableAutoStopRequestExt`
 - * `EDMA_TcdEnableInterrupts` -> `EDMA_TcdEnableInterruptsExt`
 - * `EDMA_TcdDisableInterrupts` -> `EDMA_TcdDisableInterruptsExt`
 - * `EDMA_TcdSetMajorOffsetConfig` -> `EDMA_TcdSetMajorOffsetConfigExt`

[2.9.2]

- Improvements
 - Remove `tcd` alignment check in API that is low level and does not necessarily use scatter/gather mode.

[2.9.1]

- Bug Fixes
 - Deinit channel request source before set channel mux.

[2.9.0]

- Improvements
 - Release peripheral from reset if necessary in init function.
- Bug Fixes
 - Fixed the variable type definition error issue.
 - Fixed doxygen warning.
 - Fixed violations of MISRA C-2012 rule 18.1.

[2.8.1]

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

[2.8.0]

- Improvements
 - Added feature FSL_FEATURE_EDMA_HAS_NO_CH_SBR_SEC to separate DMA without SEC bitfield.

[2.7.1]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.3, 10.4, 11.6, 11.8, 14.3,.

[2.7.0]

- Improvements
 - Use more accurate DMA instance based feature macros.
- New Features
 - Add new APIs EDMA_PrepareTransferTCD and EDMA_SubmitTransferTCD, which support EDMA transfer using TCD.

[2.6.0]

- Improvements
 - Modify the type of parameter channelRequestSource from dma_request_source_t to int32_t in the EDMA_SetChannelMux.

[2.5.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.3, 10.4, 11.6, 20.7, 12.2, 20.9, 5.3, 10.8, 8.4, 9.3.

[2.5.2]

- Improvements
 - Applied ERRATA 51327.

[2.5.1]

- Bug Fixes
 - Fixed the EDMA_ResetChannel function cannot reset channel DONE/ERROR status.

[2.5.0]

- Improvements
 - Added feature FSL_FEATURE_EDMA_HAS_NO_SBR_ATTR_BIT to separate DMA without ATTR bitfield.
 - Added api EDMA_GetChannelSystemBusInformation to gets the channel identification and attribute information on the system bus interface.
- Bug Fixes
 - Fixed the ESG bit not set in scatter gather mode issue.

- Fixed the DREQ bit configuration missed in single transfer issue.
- Cleared the interrupt status before invoke callback to avoid miss interrupt issue.
- Removed `disableRequestAfterMajorLoopComplete` from `edma_transfer_config_t` structure as driver will handle it.
- Fixed the channel mux configuration not compatible issue.
- Fixed the out of bound access in function `EDMA_DriverIRQHandler`.

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.2]

- Improvements
 - Fixed HIS ccm issue in function `EDMA_PrepareTransferConfig`.
 - Fixed violations of MISRA C-2012 rule 11.6, 10.7, 10.3, 18.1.
- Bug Fixes
 - Added ACTIVE & BITER & CITER bitfields to determine the channel status to fixed the issue of the transfer request cannot submit by function `EDMA_SubmitTransfer` when channel is idle.

[2.3.1]

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

[2.3.0]

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

[2.2.0]

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

[2.1.9]

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

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

ENET**[2.11.0]**

- New Features
 - Added function ENET_Ptp1588JumpTimer which adjusts the ENET PTP 1588 timer by jumping a relative time difference. Compared to ENET_Ptp1588SetTimer, this function yields more accurate results when the relative time difference between the PTP clock and the target clock is known.

[2.10.1]

- Bug Fixes
 - Fixed WAKEUP interrupt not being handled.

[2.10.0]

- New Features
 - Added function `ENET_Ptp1588GetChannelCaptureValue` to read last captured time from PTP 1588 timer.

[2.9.3]

- Bug Fixes
 - Fixed `ENET_Ptp1588GetTimer` incorrect timestamps when timer wraps occur after nanosecond capture:
 - * Now only increments second field when nanosecond value is less than half a second

[2.9.2]

- Bug Fixes
 - RGMII mode is (temporarily) disabled before selecting between 10/100-Mbit/s and 1000-Mbit/s modes of operation. The bit `RGMII_EN` of RCR register must not be set while changing ECR register's speed bit, otherwise there is a possibility of ENET IP ending in an incorrect state.

[2.9.1]

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

[2.9.0]

- Bug Fixes
 - Enabled collection of transfer statistics, so the function `ENET_GetStatistics` does not always return zeroes.
- New Features
 - Added new function `ENET_EnableStatistics` to enable/disable collection of transfer statistics.
 - Added new function `ENET_ResetStatistics` to reset transfer statistics.
- Improvements
 - Renamed the function `ENET_ResetHareware` to `ENET_ResetHardware`.

[2.8.0]

- New Features
 - Added the function to reset hardware on certain devices.

[2.7.1]

- Bug Fixes
 - Fixed the issue that free wrong buffer address when one frame stores in multiple buffers and memory pool is not enough to allocate these buffers to receive one complete frame.

[2.7.0]

- Improvements
 - Deleted deprecated zero copy Tx/Rx functions and set callback function which can be configured in ENET_Init.
 - Moved the Rx zero copy buffer allocation to Rx BD initialization function to reduce unnecessary looping code.
- Bug Fixes
 - Fixed the issue that predefined Rx buffers which should not be used when enabling Rx zero copy are still be handled by cache operation, it causes hardfault on some platforms.
 - Fixed the issue that zero-copy Rx function doesn't check Rx length of 0 in the BD with EMPTY bit is 0, it may occur in the corner case reported by customer. Not sure how it turns out, consider it as an ENET IP issue and drop this abnormal BD.

[2.6.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 11.6.

[2.6.2]

- Improvements
 - Changed ENET1_MAC0_Rx_Tx_Done0_DriverIRQHandler/ENET1_MAC0_Rx_Tx_Done1_DriverIRQHandler to ENET1_MAC0_Rx_Tx_Done1_DriverIRQHandler/ENET1_MAC0_Rx_Tx_Done2_DriverIRQHandler which represent ring 1 and ring 2.

[2.6.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3, 10.4, 10.7, 11.6, 11.8.

[2.6.0]

- Improvements
 - Added MDIO access wrapper APIs for ease of use.
 - Fixed the build warning introduced by 64-bit compatibility patch.

[2.5.4]

- Improvements
 - Made the driver compatible with 64-bit platforms.

[2.5.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 11.6.

[2.5.2]

- Improvements
 - Updated the TXIC/RXIC register handling code according to the new header file.

[2.5.1]

- Bug Fixes
 - Fixed document typo.

[2.5.0]

- Bug Fixes
 - Fixed the SendFrame/SendFrameZeroCopy functions issue with scattered buffers.
 - Updated the formula of MDC calculation.
 - Used a feature macro to distinguish the old IP design from the new design, because old IP design always reads a value zero from ATCR->CAPTURE bit. For old IP, driver calculates and wait the necessary delay cycles after setting ATCR->CAPTURE then gets the timestamp value.
- New Features
 - Added new zero copy Tx/Rx function.
 - New zero copy Tx function combines scattered and contiguous Tx buffer in one API, it also supports more Tx features which buffer descriptor supports but previous Tx function doesn't support.
 - New zero copy Rx function use dynamic buffer mechanism and simpler interface.
- Improvements
 - Corrected the interrupt handler for PTP timestamp IRQ and PTP1588 event IRQ since platform difference.
 - Added missing IRQ handlers for PTP1588 events on some platforms.
 - Corrected the max Tx frame length verification, it will not depend on a fixed macro. The ENET_FRAME_MAX_FRAMELEN is only an default value for driver, application can configure it. Driver calculates the limitation with the max frame length in register which may takes extended 4 or 8 bytes VLAN tag if VLAN/SVLAN enables.
 - Deleted deprecated Clause 45 read/write legacy APIs.

[2.4.3]

- Improvements
 - Aligned the IRQ handler name with header file.

[2.4.2]

- Bug Fixes
 - Fixed the MISRA issue of speculative out-of-bounds access.

[2.4.1]

- Bug Fixes
 - Fixed the PTP time capture issue.

[2.4.0]

- Improvements
 - Exposed API ENET_ReclaimTxDescriptor for user application to reclaim tx descriptors in their application.
 - Added counter to record multicast hash conflict in struct _enet_handle, improved the situation that one multicast group could be left by other conflict multicast address left operation.
 - Improved concurrent usage of reclaim and send frame operation.

[2.3.4]

- Bug Fixes
 - Fixed the issue that interrupt handler only checks the interrupt event flag but not checks interrupt mask flag.

[2.3.3]

- Bug Fixes
 - Fixed the issue that some compilers may choose the memcpy with 4-bit aligned address limitation due to the type of address pointer is 'unsigned int *', the data address doesn't have to be 4-bit aligned.

[2.3.2]

- New Features
 - Added the feature that ENET driver can be used in the platform which integrates both 10/100M and 1G ENET IP.
 - Deleted duplicated code about ARM errata 838869 in first/second level IRQ handler.

[2.3.1]

- Improvements
 - Added function pointer checking in IRQ handler to make sure code can be used even it runs into the interrupt when the second level interrupt handler is NULL.

[2.3.0]

- Bug Fixes
 - Fixed the issue that clause 45 MDIO read/write API doesn't check the transmission over status between two transmissions.
 - Fixed violations of the MISRA C-2012 rules 2.2,10.3,10.4,10.7,11.6,11.8,13.5,14.4,15.7,17.7.
- New Features
 - Added APIs to support send/receive frame with Zero-Copy.
- Improvements
 - Separated the clock configuration from module configuration when init and deinit.
 - Added functions to set second level interrupt handler.
 - Provided new function to get 1588 timer count without disabling interrupt.
 - Improved timestamp controlling, deleted all old timestamp management APIs and data structures.
 - Merged the single/multiple ring(s) APIs, now these APIs can handle both.
 - Used base and index to control buffer descriptor, aligned with qos and lpc enet driver.

[2.2.6]

- Bug Fixes
 - Updated MII speed formula referring to the manual.

[2.2.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.7, 11.6, 11.9, 13.5, 14.4, 16.4, 17.7, 21.15, 3.1, 8.4.
 - Changed to use ARRAY_SIZE(s_enetBases) as the array size for s_ENETHandle, fixed the hardfault issue for using some ENET instance when ARRAY_SIZE(s_enetBases) is not same as FSL_FEATURE_SOC_ENET_COUNT.

[2.2.4]

- Improvements
 - Added call to Data Synchronization Barrier instruction before activating Tx/Rx buffer descriptor to ensure previous data update is completed.
 - Improved ENET_TransmitIRQHandler to store timestamps for multiple transmit buffer descriptors.
- Bug Fixes
 - Fixed the issue that ENET_Ptp1588GetTimer did not handle the timer wrap situation.

[2.2.3]

- Improvements
 - Improved data buffer cache maintenance in the ENET driver.

[2.2.2]

- New Features
 - Added APIs for extended multi-ring support.
 - Added the AVB configure API for extended AVB feature support.

[2.2.1]

- Improvements
 - Changed the input data pointer attribute to const in ENET_SendFrame().

[2.1.1]

- New Features
 - Added the extended MDIO IEEE802.3 Clause 45 MDIO format SMI command APIs.
 - Added the extended interrupt coalescing feature.
- Improvements
 - Combined all storage operations in the ENET_Init to ENET_SetHandler API.

[2.0.1]

- Bug Fixes
 - Used direct transmit busy check when doing data transmit.
- Miscellaneous Changes
 - Updated IRQ handler work flow.
 - Changed the TX/RX interrupt macro from kENET_RxByteInterrupt to kENET_RxBufferInterrupt, from kENET_TxByteInterrupt to kENET_TxBufferInterrupt.
 - Deleted unnecessary parameters in ENET handler.

[2.0.0]

- Initial version.
-

ENET_QOS

[2.7.2]

- Bug Fixes
 - Not disabling ENET_QOS_ConfigureRxParser function when compiled for EMAC, as the MTL_RXP_INDIRECT_ACC_DATA register is writeable now.

[2.7.1]

- Bug Fixes
 - Fixed writing after DMA_CH array on platforms with smaller number of channels (EMAC).

[2.7.0]

- New features
 - The driver can work with EMAC IP now. EMAC is similar to ENET_QOS but doesn't support 100 % of its features.

[2.6.5]

- Bug Fixes
 - Fixed ENET_QOS_GetMacAddr address byte order not matching ENET_QOS_SetMacAddr.

[2.6.4]

- Improvements
 - ENET_QOS_SetMII returns success or failure status now (related to i.MX RT1170 errata ERR050539 - ENET_QOS doesn't support RMII 10Mbps mode).
- Bug Fixes
 - Fixed the MISRA C-2012 issue rule 14.3.

[2.6.3]

- Bug Fixes
 - Fixed the issue that ENET_QOS_GetRxFram, ENET_QOS_ReadFrame and ENET_QOS_DropFrame did not properly restart the receiving once it stopped.

[2.6.2]

- Bug Fixes
 - Fixed the issue that free wrong buffer address when one frame stores in multiple buffers and memory pool is not enough to allocate these buffers to receive one complete frame.

[2.6.1]

- Bug Fixes
 - Fixed the issue that ENET_QOS_ReadFrame doesn't check timestamp available bit before check the context BD bit, it makes software update extra BD. If DMA receives new frame to this BD before software update, software will lose this frame.

[2.6.0]

- New features
 - Added hardware checksum acceleration support.

[2.5.3]

- Bug Fixes
 - Fixed the MISRA issue rule 14.3, 5.3.

[2.5.2]

- Bug Fixes
 - Fixed the issue that ENET_QOS_Init reset the MDIO setting of ENET_QOS_SetSML.

[2.5.1]

- Improvements
 - Supported RMII mode.

[2.5.0]

- Improvements
 - Added MDIO access wrapper APIs for ease of use.

[2.4.1]

- Improvements
 - Supported cache control.
 - Supported BD address conversion to system address.
 - Make driver aarch64 compatible
- Bug Fixes
 - Fixed the issue that driver internal interface ENET_QOS_DropFrame drops all frames in whole BD ring rather than one frame as design. Impact case: 1. Rx drop occurs in zero copy Rx API ENET_QOS_GetRxFrame. 2. Call ENET_QOS_ReadFrame with data pointer is NULL, driver will drop all Rx frames.

[2.4.0]

- New features
 - Added MDIO IEEE802.3 Clause 45 access support.
 - Added get statistics API to get some statistical data in transfer.
 - Added new APIs to support zero copy Rx.
 - Fixed the MISRA issue rule 8.4, 8.6.

[2.3.0]

- Improvements
 - Added counter to record multicast hash conflict in struct _enet_handle, improved the situation that one multicast group could be left by other conflict multicast address left operation.
- Bug Fixes
 - Updated txDirtyRing maintenance in reclaim and send frame process, allow txDirtyRing to be overwritten.
 - Disabled carrier sensing in full duplex mode configuration in ethernet initialization
 - Fixed 1588 sub-second calculate issue.

[2.2.2]

- Bug Fixes
 - Fixed the issue that ENET_QOS_SetupTxDescriptor didn't handle the DMA access address mapping for SoCs have feature FSL_FEATURE_MEMORY_HAS_ADDRESS_OFFSET.
 - Fixed MISRA 2012 violations detected in examples build.

[2.2.1]

- Bug Fixes
 - Fixed MISRA 2012 violations, fixed doxygen warning.
 - Fixed the issue that cache invalidate to invalid converted memory address in ENET_QOS_ReadFrame for SoCs have feature FSL_FEATURE_MEMORY_HAS_ADDRESS_OFFSET.

[2.2.0]

- Removed the ptp time data ring management, below structures and APIs are removed:
 - structure enet_qos_ptp_time_data_t
 - structure enet_qos_ptp_time_data_ring_t
 - API ENET_QOS_GetRxFrameTime
 - API ENET_QOS_GetTxFrameTime
- Added API for GCL list read and AVB configuration
 - ENET_QOS_EstReadGcl
 - ENET_QOS_AVBConfigure
- Improved driver for PTP system time configuration, timestamp read.
- Added IRQ lock and memory barrier instruction for descriptor operation.
- Fixed MISRA 2012 violations

[2.1.1]

- Bug Fixes
 - Fixed the bug that data pointer is not converted to local memory address in the call to ENET_QOS_Ptp1588ParseFrame.

[2.1.0]

- New feature
 - Update driver to support feature FSL_FEATURE_MEMORY_HAS_ADDRESS_OFFSET which convert buffer address to visible address for DMA.
 - Require user to provide implementation for ENET_QOS_SetSYSControl API, which set the PHY interface and enable clock generation for IP.

[2.0.0]

- Initial version.

FLEXCAN

[2.14.5]

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

[2.14.4]

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

[2.14.3]

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

[2.14.2]

- Improvements
 - Add Coverage Justification for uncovered code.
 - Adjust API FLEXCAN_TransferAbortReceive order.
 - Update FLEXCAN_Enable to enter Freeze Mode first when enter Disable mode on some platform.
 - Added while loop timeout for following API:
 - * FLEXCAN_EnterFreezeMode
 - * FLEXCAN_ExitFreezeMode
 - * FLEXCAN_Enable
 - * FLEXCAN_Reset
 - * FLEXCAN_TransferSendBlocking
 - * FLEXCAN_TransferReceiveBlocking
 - * FLEXCAN_TransferFDSendBlocking
 - * FLEXCAN_TransferFDReceiveBlocking

- * FLEXCAN_TransferReceiveFifoBlocking
- * FLEXCAN_TransferReceiveEnhancedFifoBlocking
- Bug Fixes
 - Remove remote frame feature in CANFD mode because there is no remote frame in the CANFD format.
 - Remove legacy Rx FIFO disabled branch in FLEXCAN_SubHandlerForLegacyRxFIFO and FLEXCAN_SubHandlerForDataTransferred.

[2.14.1]

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

[2.14.0]

- Improvements
 - Support external time tick feature.
 - Support high resolution timestamp feature.
 - Enter Freeze Mode first when enter Disable Mode on some platform.
 - Add feature macro for Pretended Networking because some FlexCAN instance do not have this feature.
 - Add feature macro for enhanced Rx FIFO because some FlexCAN instance do not have this feature.
 - Add new FlexCAN IRQ Handler FLEXCAN_DriverDataIRQHandler and FLEXCAN_DriverEventIRQHandler. These IRQ Handlers are used on soc which FlexCAN interrupts are grouped by specific function and assigned to different vector.
 - Update macro FLEXCAN_WAKE_UP_FLAG and FLEXCAN_PNWAKE_UP_FLAG to simplify code.
 - Replace macro FSL_FEATURE_FLEXCAN_HAS_NO_WAKMSK_SUPPORT with FSL_FEATURE_FLEXCAN_HAS_NO_SLFWAK_SUPPORT.

- Replace macro `FSL_FEATURE_FLEXCAN_HAS_NO_WAKSRC_SUPPORT` with `FSL_FEATURE_FLEXCAN_HAS_GLITCH_FILTER`.
- Bug Fixes
 - Fixed wrong interrupt and status flag helper macro in enumeration `_flexcan_flags` and API `FLEXCAN_DisableInterrupts`.
 - Fixed interrupt flag helper macro typo issue.
 - Remove flags which will be unassociated with interrupt in macro `FLEXCAN_MEMORY_ERROR_INT_FLAG`.
 - Remove flags which will be unassociated with interrupt in macro `FLEXCAN_ERROR_AND_STATUS_INT_FLAG`.
 - Fixed array out-of-bounds access when read enhanced Rx FIFO.

[2.13.1]

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

[2.13.0]

- Improvements
 - Support payload endianness selection feature.

[2.12.0]

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

[2.11.8]

- Improvements
 - Synchronize flexcan driver update on s32z platform.

[2.11.7]

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

[2.11.6]

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

[2.11.5]

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

[2.11.4]

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

[2.11.3]

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

[2.11.2]

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

[2.11.1]

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

[2.11.0]

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

[2.10.1]

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

[2.10.0]

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

[2.9.2]

- Bug Fixes
 - Fixed the issue that FLEXCAN_CheckUnhandleInterruptEvents() can't detecting the exist enhanced RX FIFO interrupt status.
 - Fixed the issue that FLEXCAN_ReadPNWakeUpMB() does not return fail even no existing valid wake-up frame.
 - Fixed the issue that FLEXCAN_ReadEnhancedRxFifo() may clear bits other than the data available bit.
 - Fixed violations of the MISRA C-2012 rules 10.4, 10.8.
- Improvements
 - Return kStatus_FLEXCAN_RxFifoDisabled instead of kStatus_Fail when read FIFO fail during IRQ handler.
 - Remove unreachable code from timing calculates APIs.
 - Update Enhanced Rx FIFO handler to make it deal with underflow/overflow status first.

[2.9.1]

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

[2.9.0]

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

[2.8.7]

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

[2.8.6]

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

[2.8.5]

- Improvements
 - Make driver aarch64 compatible.

[2.8.4]

- Bug Fixes
 - Fixed FlexCan_Errata_6032 to disable all interrupts.

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.1]

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

[2.7.0]

- Improvements
 - Update config configuration.
 - * Added enablePretendedNetworking member to support enable/disable Pretended Networking feature.
 - * Added enableTransceiverDelayMeasure member to support enable/disable Transceiver Delay MeasurementPretended feature.
 - * Added bitRate/bitRateFD member to work as baudRate/baudRateFD member union.
 - Rename all “baud” in code or comments to “bit” to align with the CAN spec.
 - Added Pretended Networking mode related APIs.
 - * FLEXCAN_SetPNConfig
 - * FLEXCAN_GetPNMatchCount
 - * FLEXCAN_ReadPNWakeUpMB
 - Added support for Enhanced Rx FIFO.
 - Removed independent memory error interrupt/status APIs and put all interrupt/status control operation into FLEXCAN_EnableInterrupts/FLEXCAN_DisableInterrupts and FLEXCAN_GetStatusFlags/FLEXCAN_ClearStatusFlags APIs.
 - Update improved timing APIs to make it calculate improved timing according to CiA doc recommended.
 - * FLEXCAN_CalculateImprovedTimingValues.
 - * FLEXCAN_FDCalculateImprovedTimingValues.
 - Update FLEXCAN_SetBitRate/FLEXCAN_SetFDBitRate to added the use of enhanced timing registers.

[2.6.2]

- Improvements
 - Add CANFD frame data length enumeration.

[2.6.1]

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

[2.6.0]

- Improvements
 - Enable CANFD ISO mode in FLEXCAN_FDInit API.
 - Enable the transceiver delay compensation feature when enable FD operation and set bitrate switch.
 - Implementation memory error control in FLEXCAN_Init API.
 - Improve FLEXCAN_FDCalculateImprovedTimingValues API to get same value for FPRESDIV and PRES DIV.
 - Added memory error configuration for user.

- * enableMemoryErrorControl
- * enableNonCorrectableErrorEnterFreeze
- Added memory error related APIs.
 - * FLEXCAN_GetMemoryErrorReportStatus
 - * FLEXCAN_GetMemoryErrorStatusFlags
 - * FLEXCAN_ClearMemoryErrorStatusFlags
 - * FLEXCAN_EnableMemoryErrorInterrupts
 - * FLEXCAN_DisableMemoryErrorInterrupts
- Bug Fixes
 - Fixed the issue of sent duff CAN frame after call FLEXCAN_FDInit() API.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.7, rule-10.8, rule-11.8, rule-12.2, rule-13.4, rule-14.4, rule-15.5, rule-15.6, rule-15.7, rule-16.4, rule-17.3, rule-5.8, rule-8.3, rule-8.5.
 - Fixed the issue that API FLEXCAN_SetFDRxMbConfig lacks inactive message buff.
 - Fixed the issue of Pa082 warning.
 - Fixed the issue of dead lock in the function of interruption handler.
 - Fixed the issue of Legacy Rx Fifo EDMA transfer data fail in evkmimxrt1060 and evk-mimxrt1064.
 - Fixed the issue of setting CANFD Bit Rate Switch.

- Fixed the issue of operating unknown pointer risk.
 - * when used the pointer “handle->mbFrameBuf[mbIdx]” to update the timestamp in a short-live TX frame, the frame pointer became as unknown, the action of operating it would result in program stack destroyed.
- Added assert to check current CAN clock source affected by other clock gates in current device.
 - * In some chips, CAN clock sources could be selected by CCM. But for some clock sources affected by other clock gates, if user insisted on using that clock source, they had to open these gates at the same time. However, they should take into consideration the power consumption issue at system level. In RT10xx chips, CAN clock source 2 was affected by the clock gate of lpuart1. ERRATA ID: (ERR050235 in CCM).
- Improvements
 - Implementation for new FLEXCAN with ECC feature able to exit Freeze mode.
 - Optimized the function of interruption handler.
 - Added two APIs for FLEXCAN EDMA driver.
 - * FLEXCAN_PrepareTransfConfiguration
 - * FLEXCAN_StartTransferDatafromRxFIFO
 - Added new API for FLEXCAN driver.
 - * FLEXCAN_GetTimeStamp
 - For TX non-blocking API, we wrote the frame into mailbox only, so no need to register TX frame address to the pointer, and the timestamp could be updated into the new global variable handle->timestamp[mbIdx], the FLEXCAN driver provided a new API for user to get it by handle and index number after TX DONE Success.
 - * FLEXCAN_EnterFreezeMode
 - * FLEXCAN_ExitFreezeMode
 - Added new configuration for user.
 - * disableSelfReception
 - * enableListenOnlyMode
 - Renamed the two clock source enum macros based on CLKSRC bit field value directly.
 - * The CLKSRC bit value had no property about Oscillator or Peripheral type in lots of devices, it acted as two different clock input source only, but the legacy enum macros name contained such property, that misled user to select incorrect CAN clock source.
 - Created two new enum macros for the FLEXCAN driver.
 - * kFLEXCAN_ClkSrc0
 - * kFLEXCAN_ClkSrc1
 - Deprecated two legacy enum macros for the FLEXCAN driver.
 - * kFLEXCAN_ClkSrcOsc
 - * kFLEXCAN_ClkSrcPeri
 - Changed the process flow for Remote request frame response..
 - * Created a new enum macro for the FLEXCAN driver.
 - kStatus_FLEXCAN_RxRemote

- Changed the process flow for kFLEXCAN_StateRxRemote state in the interrupt handler.
 - * Should the TX frame not register to the pointer of frame handle, interrupt handler would not be able to read the remote response frame from the mail box to ram, so user should read the frame by manual from mail box after a complete remote frame transfer.

[2.4.0]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-12.1, rule-17.7, rule-16.4, rule-11.9, rule-8.4, rule-14.4, rule-10.8, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-11.6, rule-13.5, rule-11.3, rule-8.3, rule-12.2 and rule-16.1.
 - Fixed the issue that CANFD transfer data fail when bus baudrate is 30Khz.
 - Fixed the issue that ERR009595 does not follow the ERRATA document.
 - Fixed code error for ERR006032 work around solution.
 - Fixed the Coverity issue of BAD_SHIFT in FLEXCAN.
 - Fixed the Repo build warning issue for variable without initial.
- Improvements
 - Fixed the run fail issue of FlexCAN RemoteRequest UT Case.
 - Implementation all TX and RX transferring Timestamp used in FlexCAN demos.
 - Fixed the issue of UT Test Fail for CANFD payload size changed from 64BperMB to 8PerMB.
 - Implementation for improved timing API by baud rate.

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

FLEXCAN_EDMA**[2.12.1]**

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

[2.12.0]

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

[2.11.7]

- Refer FLEXCAN driver change log 2.7.0 to 2.11.7
-

FLEXIO

[2.3.0]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Improvements
 - Added doxygen index parameter comment in FLEXIO_SetClockMode.

[2.2.0]

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

[2.1.0]

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

[2.0.4]

- Bug Fixes
 - Fixed MISRA 8.4 issues.

[2.0.3]

- Bug Fixes
 - Fixed MISRA 10.4 issues.

[2.0.2]

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

[2.0.1]

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

FLEXIO_I2C**[2.6.2]**

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues

[2.6.0]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_I2S

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_I2S_EDMA**[2.1.9]**

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

[2.1.8]

- Improvements
 - Applied EDMA ERRATA 51327.
-

FLEXIO_SPI

[2.4.3]

- Improvements
 - Make SPI_RETRY_TIMES configurable by CONFIG_SPI_RETRY_TIMES.

[2.4.2]

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

[2.4.1]

- Bug Fixes
 - Fixed coverity issues

[2.4.0]

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

[2.3.5]

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

[2.3.4]

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

[2.3.3]

- Bugfixes
 - Fixed cs-continuous mode.

[2.3.2]

- Improvements
 - Changed FLEXIO_SPI_DUMMYDATA to 0x00.

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.3]

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

[2.1.2]

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

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_UART

[2.6.4]

- Improvements
 - Make UART_RETRY_TIMES configurable by CONFIG_UART_RETRY_TIMES.

[2.6.3]

- Bug Fixes
 - Fixed coverity issues

[2.6.2]

- Bug Fixes
 - Fixed coverity issues

[2.6.1]

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

[2.6.0]

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

[2.5.1]

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

[2.5.0]

- Improvements
 - Added API FLEXIO_UART_FlushShifters to flush UART fifo.

[2.4.0]

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

[2.3.0]

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

[2.2.0]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

FLEXIO_UART_EDMA

[2.3.1]

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

[2.3.0]

- Refer FLEXIO_UART driver change log to 2.3.0
-

FLEXSPI

[2.8.1]

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

[2.8.0]

- Bug Fixes
 - Introduced the **disableAhbReadResume** field in the flexspi_config_t structure to provide control over the AHBCR[RESUMEDISABLE] register bit.
 - Implemented a workaround for hardware erratum ERR052733 by setting the default value of **disableAhbReadResume** to **true**.
 - Fixed issue in FLEXSPI_TransferHandleIRQ where the transfer completion was incorrectly signaled despite pending read/write operations.
- New Features
 - Introduced a new function(FLEXSPI_UpdateAhbBuffersSettings) that allows users to update the AHB buffer configuration after the FLEXSPI module has been initialized

[2.7.0]

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

[2.6.4]

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

[2.6.3]

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

[2.6.2]

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

[2.6.1]

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

[2.6.0]

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

[2.5.1]

- Bugfixes
 - Fixed handling of W1C bits in the INTR register
 - Removed FIFO resets from FLEXSPI_CheckAndClearError
 - FLEXSPI_TransferBlocking is observing IPCMDDONE and then fetches the final status of the transfer
 - Fixed issue that FLEXSPI2_DriverIRQHandler not defined.

[2.5.0]

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

[2.4.0]

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

[2.3.5]

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

[2.3.4]

- Bug Fixes
 - Updated flexspi_config_t structure and FlexSPI_Init to support new feature FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_CONBINATION.

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

- Improvements
 - Defaulted enable prefetch for AHB RX buffer configuration in `FLEXSPI_GetDefaultConfig`, which is align with the reset value in `AHBRXBUFxCR0`.
 - Added software workaround for ERR011377 in `FLEXSPI_SetFlashConfig`; added some delay after DLL lock status set to ensure correct data read/write.

[2.1.0]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

FLEXSPI EDMA Driver**[2.3.3]**

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

[2.3.2]

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

[2.0.2]

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

[2.0.0]

- Initial version.
-

I3C

[2.14.3]

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

[2.14.2]

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

[2.14.1]

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

[2.14.0]

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

[2.13.1]

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

[2.13.0]

- New features
 - Added the hot-join support for I3C bus initialization API.
- Bug Fixes
 - Set read termination with START at the same time in case unknown issue.
 - Set MCTRL[TYPE] as 0 for DDR force exit.
- Improvements
 - Added the API to reset device count assigned by ENTDAAs.
 - Provided the method to set global macro I3C_MAX_DEVCNT to determine how many device addresses ENTDAAs can allocate at one time.
 - Initialized target management static array based on instance number for the case that multiple instances are used at the same time.

[2.12.0]

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

[2.11.1]

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

[2.11.0]

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

[2.10.8]

- Improvements
 - Support more instances.

[2.10.7]

- Improvements
 - Fixed the potential compile warning.

[2.10.6]

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

[2.10.5]

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

[2.10.4]

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

[2.10.3]

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

[2.10.2]

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

[2.10.1]

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

[2.10.0]

- New features
 - Added I3C extended IBI data support.

[2.9.0]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.4]

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

[2.5.3]

- Updated driver for new feature FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH and FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

- Improvements
 - Added I3C common driver APIs to initialize I3C with both master and slave configuration.
 - Updated I3C master transfer callback to function set structure to include callback invoke for IBI event and slave2master event.
 - Updated I3C master non-blocking transfer model and always enable the interrupts to be able to re-act to the slave start event and handle slave IBI.

[2.2.0]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

I3C_EDMA

[2.2.12]

- Bug Fixes
 - Fixed the issue that EDMA transfer configuration use wrong parameter.

[2.2.11]

- Improvements
 - Fixed Coverity CERT-C violations.

[2.2.10]

- Bug Fixes
 - Fixed the issue that slave start event is cleared when it has not been handled.
- Added
 - Supported I3C HDR-DDR transfer with EDMA.
- Changed
 - Used linked EDMA to transfer all I3C subaddress and data without handling of intermediate states, simplifying code logic.
 - Prepare DMA before I3C START to ensure there's no time delay between START and transmitting data.
 - Added the MCTRLDONE flag check after START and STOP request to ensure all states are handled properly.

[2.2.9]

- Bug Fixes
 - Fixed MISRA issue rule 11.3.
 - Added the master control done flag waiting code after STOP in case the bus is not idle when transfer function finishes.

[2.2.8]

- Improvements
 - Removed I3C IRQ handler calling in the EDMA callback. Previously driver doesn't use the END byte which can trigger the STOP interrupt for controller sending and receiving, now let I3C event handler deal with all I3C events.
- Bug Fixes
 - Fixed the bug that the END type Tx register is not used when command length or data length is one byte.

[2.2.7]

- Bug Fixes
 - Fixed MISRA issue rule 11.6.

[2.2.6]

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

[2.2.5]

- Improvements
 - Added the workaround for RT1180 I3C EDMA issue ERR052086.

[2.2.4]

- Bug Fixes
 - Fixed the issue that I3C master sends the last byte data without using the END type register.

[2.2.3]

- Bug Fixes
 - Fixed issue that slave pollutes the last byte when Tx FIFO may be full.

[2.2.2]

- Bug Fixes
 - Fixed I3C MISRA issue rule 10.4, 11.3.

[2.2.1]

- Bug Fixes
 - Fixed the issue that I3C slave send the last byte data without using the END type register.
- Improvements
 - There's no need to reserve two bytes FIFO for DMA transfer which is for IP issue workaround.

[2.2.0]

- Improvements
 - Deleted legacy IBI data request code.

[2.1.0]

- Bug Fixes
 - Fixed MISRA issue rule 8.4, 8.6, 11.8.

[2.0.1]

- Bug Fixes
 - Fixed MISRA issue rule 9.1.

[2.0.0]

- Initial version.
-

LPI2C

[2.6.2]

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

[2.6.1]

- Bug Fixes
 - Fixed coverity issues.

[2.6.0]

- New Feature
 - Added common IRQ handler entry LPI2C_DriverIRQHandler.

[2.5.7]

- Improvements
 - Added support for separated IRQ handlers.

[2.5.6]

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

[2.5.5]

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

[2.5.4]

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

[2.5.3]

- Improvements
 - Added handler for LPI2C7 and LPI2C8.

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.1]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

- Improvements
 - Updated LPI2C_GetCyclesForWidth to add the parameter of minimum cycle, because for master SDA/SCL filter, master bus idle/pin low timeout and slave SDA/SCL filter configuration, 0 means disabling the feature and cannot be used.
- Bug Fixes

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

[2.3.0]

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

[2.2.0]

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

[2.1.12]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.1.11]

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

[2.1.10]

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

[2.1.9]

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

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

LPI2C_EDMA

[2.4.5]

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

[2.4.4]

- Improvements
 - Added support for 2KB data transfer

[2.4.3]

- Improvements
 - Added support for separated IRQ handlers.

[2.4.2]

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

[2.4.1]

- Refer LPI2C driver change log 2.0.0 to 2.4.1
-

LPIT

[2.1.3]

- Bug Fixes
 - Fixed doxygen generation warnings.

[2.1.2]

- Bug Fixes
 - Fix CERT INT31-C issues.

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPSPi**[2.7.3]**

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

[2.7.2]

- Bug Fixes
 - Fixed coverity issues.

[2.7.1]

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

[2.7.0]

- New Feature
 - Added common IRQ handler entry LPSPi_DriverIRQHandler.

[2.6.10]

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

[2.6.9]

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

[2.6.8]

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

[2.6.7]

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

[2.6.6]

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

[2.6.5]

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

[2.6.4]

- Bug Fixes
 - Added LPSPi6_DriverIRQHandler for LPSPi6 instance.

[2.6.3]

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

[2.6.2]

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

[2.6.1]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPSPI_EDMA

[2.4.9]

- Improvements
 - Removed unused code from LPSPI_SeparateEdmaReadData().

[2.4.8]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

- Improvements
 - Add the TXMSK wait after TCR setting.

[2.4.0]

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

LPTMR**[2.2.1]**

- Bug Fixes
 - Fix CERT INT31-C issues.

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPUART

[2.10.0]

- New Feature
 - Added support to configure RTS watermark.

[2.9.4]

- Improvements
 - Merged duplicate code.

[2.9.3]

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

[2.9.2]

- Bug Fixes
 - Fixed coverity issues.

[2.9.1]

- Bug Fixes
 - Fixed coverity issues.

[2.9.0]

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

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.7]

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

[2.7.6]

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

[2.7.5]

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

[2.7.4]

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

[2.7.3]

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

[2.7.2]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.1]

- Bug Fixes
 - Fixed MISRA advisory 17.7 issues.

[2.4.0]

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

[2.3.1]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.3.0]

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

[2.2.8]

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

[2.2.7]

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

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - Update transactional APIs.
-

LPUART_EDMA

[2.4.0]

- Refer LPUART driver change log 2.1.0 to 2.4.0
-

MCM

[2.2.0]

- Improvements
 - Support platforms with less features.

[2.1.0]

- Others
 - Remove byteID from mcm_lmem_fault_attribute_t for document update.

[2.0.0]

- Initial version.
-

MIPI_DSI

[2.1.0]

- Bug Fixes
 - Fixed typo in member of `dsi_transfer_t` structure. The `sendDscCmd` and `dscCmd` shall be `sendDcsCmd` and `dcsCmd`.

[2.0.3]

- Improvements
 - Supported `DSI_ConfigDphy` api.
 - Supported `DSI_EnableVpgEnMode` to config the video mode pattern generator.

[2.0.2]

- Improvements
 - More precise calculation of the values of `m` & `n`.
 - Lookup table method to obtain DPHY-related parameters based on bandwidth.

[2.0.1]

- Bug Fixes.
 - Fixed MISRA C-2012 issues: 10.1, 10.3, 10.4, 10.8, 21.15

[2.0.0]

- Initial version.
-

MU

[2.8.1]

- Bug Fixes
 - Avoid incorrect `MU_BUSY_POLL_COUNT` macro use.

[2.8.0]

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

- Enhanced documentation across all functions to clarify timeout behavior and return values.

[2.7.0]

- New Features
 - Added API MU_GetRxStatusFlags.

[2.6.0]

- New Features
 - Added API MU_GetInterruptsPending.

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.1]

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

[2.2.0]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Fixed MISRA C-2012 issues.

[2.1.0]

- Improvements
 - Added new enum mu_msg_reg_index_t.

[2.0.0]

- Initial version.
-

OTFAD**[2.1.4]**

- Bug fixes
 - Fixed MISRA 2012 issue: 10.1.

[2.1.3]

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

[2.1.2]

- Bug fixes
 - Fixed MISRA 2012 issue: 10.4.

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

PDM

[2.9.3]

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

[2.9.2]

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

[2.9.1]

- Bug Fixes
 - Fixed the issue that the driver still enters the interrupt after disabling clock.

[2.9.0]

- Improvements
- Added feature FSL_FEATURE_PDM_HAS_DECIMATION_FILTER_BYPASS to config CTRL_2[DEC_BYPASS] field.
- Modify code to make the OSR value is not limited to 16.

[2.8.1]

- Improvements
- Added feature FSL_FEATURE_PDM_HAS_NO_DOZEN to handle nonexistent CTRL_1[DOZEN] field.

[2.8.0]

- Improvements
- Added feature FSL_FEATURE_PDM_HAS_NO_HWVAD to remove the support of hardware voice activity detector.
- Added feature FSL_FEATURE_PDM_HAS_NO_FILTER_BUFFER to remove the support of FIR_RDY bitfield in STAT register.

[2.7.4]

- Bug Fixes
 - Fixed driver can not determine the specific float number of clock divider.
 - Fixed PDM_ValidateSrcClockRate calculates PDM channel in wrong method issue.

[2.7.3]

- Improvements
- Added feature FSL_FEATURE_PDM_HAS_NO_VADEF to remove the support of VADEF bitfield in VAD0_STAT register.

[2.7.2]

- Improvements
- Added feature FSL_FEATURE_PDM_HAS_NO_MINIMUM_CLKDIV to decide whether the minimum clock frequency division is required.

[2.7.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4, 10.3, 10.1, 10.4, 14.4

[2.7.0]

- Improvements
 - Added api PDM_EnableHwvadInterruptCallback to support handle hwvad IRQ in PDM driver.
 - Corrected the sample rate configuration for non high quality mode.
 - Added api PDM_SetChannelGain to support adjust the channel gain.

[2.6.0]

- Improvements
 - Added new features FSL_FEATURE_PDM_HAS_STATUS_LOW_FREQ/FSL_FEATURE_PDM_HAS_DC_OUT

[2.5.0]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4, 16.5, 10.4, 10.3, 10.1, 11.9, 17.7, 10.6, 14.4, 11.8, 11.6.

[2.4.1]

- Bug Fixes
 - Fixed MDK 66-D warning in pdm driver.

[2.4.0]

- Improvements
 - Added api PDM_TransferSetChannelConfig/PDM_ReadFifo to support read different width data.
 - Added feature FSL_FEATURE_PDM_HAS_RANGE_CTRL and api PDM_ClearRangeStatus/PDM_GetRangeStatus for range register.
- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 14.4, 10.3, 10.4.

[2.3.0]

- Improvements
 - Enabled envelope/energy voice detect mode by adding apis PDM_SetHwvadInEnvelopeBasedMode/PDM_SetHwvadInEnergyBasedMode.
 - Added feature FSL_FEATURE_PDM_CHANNEL_NUM for different SOC.

[2.2.1]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.6, 10.7, 11.3, 11.8, 14.4, 17.7, 18.4.
 - Added medium quality mode support in function PDM_SetSampleRateConfig.

[2.2.0]

- Improvements
 - Added api PDM_SetSampleRateConfig to improve user experience and marked api PDM_SetSampleRate as deprecated.

[2.1.1]

- Improvements
 - Used new SDMA API SDMA_SetDoneConfig instead of SDMA_EnableSwDone for PDM SDMA driver.

[2.1.0]

- Improvements
 - Added software buffer queue for transactional API.

[2.0.1]

- Improvements
 - Improved HWVAD feature.

[2.0.0]

- Initial version.
-

PDM_EDMA**[2.6.5]**

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

[2.6.4]

- Improvements
 - Add handling for runtime change of number of linked transfers

[2.6.3]

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

[2.6.2]

- Improvements
 - Add macro `MCUX_SDK_PDM_EDMA_PDM_ENABLE_INTERNAL` to let the user decide whether to enable it when calling `PDM_TransferReceiveEDMA`.

[2.6.1]

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

[2.6.0]

- Improvements
 - Updated api `PDM_TransferReceiveEDMA` to support channel block interleave transfer.
 - Added new api `PDM_TransferSetMultiChannelInterleaveType` to support channel interleave type configurations.

[2.5.0]

- Refer PDM driver change log 2.1.0 to 2.5.0
-

RGPIO

[2.1.0]

- New feature:
 - Added API `RGPIO_EnablePortInput()`
 - Added API `RGPIO_SetPinInterruptConfig()`
 - Added API `RGPIO_GetPinsInterruptFlags()`
 - Added API `RGPIO_ClearPinsInterruptFlags()`

[2.0.3]

- Improvements:
 - Enhanced `FGPIO_PinInit` to enable clock internally.

[2.0.2]

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

[2.0.1]

- API Interface Change:
 - Refined naming of API while keep all original APIs with marking them as deprecated. The original API will be removed in the next release. The main change is to update API with prefix of `_PinXXX()` and `_PortXXX()`.

[2.0.0]

- Initial version.
-

SAI

[2.4.10]

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

[2.4.9]

- Added Errata ERR051421 workaround.

[2.4.8]

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

[2.4.7]

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

[2.4.6]

- Bug Fixes
 - Fixed the IAR build warning.

[2.4.5]

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

[2.4.4]

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

[2.4.3]

- Bug Fixes
 - Fixed enumeration sai_fifo_combine_t value configuration issue.

[2.4.2]

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

[2.4.1]

- Bug Fixes
 - Fixed bitWidth incorrectly assigned issue.

[2.4.0]

- Improvements
 - Removed deprecated APIs.

[2.3.8]

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

[2.3.7]

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

[2.3.6]

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

[2.3.5]

- Improvements
 - Make driver to be aarch64 compatible.

[2.3.4]

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

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

- Improvements
 - Added new APIs for parameters collection and simplified user interfaces:
 - * SAI_Init
 - * SAI_SetMasterClockConfig
 - * SAI_TxSetBitClockRate
 - * SAI_TxSetSerialDataConfig
 - * SAI_TxSetFrameSyncConfig
 - * SAI_TxSetFifoConfig
 - * SAI_TxSetBitclockConfig
 - * SAI_TxSetConfig
 - * SAI_TxSetTransferConfig

- * SAI_RxSetBitClockRate
- * SAI_RxSetSerialDataConfig
- * SAI_RxSetFrameSyncConfig
- * SAI_RxSetFifoConfig
- * SAI_RxSetBitclockConfig
- * SAI_RXSetConfig
- * SAI_RxSetTransferConfig
- * SAI_GetClassicI2SConfig
- * SAI_GetLeftJustifiedConfig
- * SAI_GetRightJustifiedConfig
- * SAI_GetTDMConfig

[2.1.9]

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

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

SAI_EDMA

[2.7.3]

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

[2.7.2]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

- Improvements
 - Removed deprecated APIs.

[2.5.1]

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

[2.5.0]

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

[2.4.0]

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

[2.3.2]

- Refer SAI driver change log 2.1.0 to 2.3.2
-

SAR_ADC**[2.3.0]**

- New Feature
 - Added new feature macro a for compatibility with ADCs on some platforms where some instances do not support group3.

[2.2.0]

- New Feature
 - Added new features to compatible with new platforms.

[2.1.1]

- Improvement
 - Change ADC sample rate phase duration default value from 0x08 to 0x14.

[2.1.0]

- New Feature
 - Added ADC_StopConvChain function to support stop scan in normal conversion scan operation mode.

[2.0.3]

- Bug Fixes
 - Fixed the array name usage error in function ADC_GetInstance.

[2.0.2]

- Bug Fixes
 - Fixed MISRA issues.

[2.0.1]

- Bug Fixes
 - Fixed the bug that when calling function ADC_EnableWdgThresholdInt() in function ADC_SetAnalogWdgConfig(), the parameter was passed incorrectly.

[2.0.0]

- Initial version.
-

SEMA42

[2.1.1]

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

[2.1.0]

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

[2.0.4]

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

[2.0.3]

- Improvements
 - Changed to implement SEMA42_Lock base on SEMA42_TryLock.

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

TMU**[2.1.0]**

- New feature.
 - Supported rising & falling temp rate critical threshold.
 - Supported polling function to avoid the temp sudden jump issue.

[2.0.0]

- Initial version.
 - This module was first developed on i.MX 93.
-

TPM**[2.4.1]**

- Improvements
 - Add Coverage Justification for uncovered code.

[2.4.0]

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

- * TPM_SetupOutputCompare
- * TPM_SetTimerPeriod
- * TPM_StopTimer

[2.3.6]

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

[2.3.5]

- New Feature
 - Added IRQ handler entry for TPM2.

[2.3.4]

- New Feature
 - Added common IRQ handler entry TPM_DriverIRQHandler.

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

- Improvements
 - Added TPM_SetChannelPolarity to support select channel input/output polarity.
 - Added TPM_EnableChannelExtTrigger to support enable external trigger input to be used by channel.
 - Added TPM_CalculateCounterClkDiv to help calculates the counter clock prescaler.
 - Added TPM_GetChannelValue to support get TPM channel value.
 - Added new TPM configuration.
 - * syncGlobalTimeBase
 - * extTriggerPolarity
 - * chnlPolarity
 - Added new PWM signal configuration.
 - * secPauseLevel
- Bug Fixes
 - Fixed TPM_SetupPwm can't configure 0% combined PWM issues.

[2.1.1]

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

[2.1.0]

- Improvements
 - Added TPM_EnableChannel and TPM_DisableChannel APIs.
 - Added new PWM signal configuration.
 - * pauseLevel - Support select output level when counter first enabled or paused.

- * enableComplementary - Support enable/disable generate complementary PWM signal.
- * deadTimeValue - Support deadtime insertion for each pair of channels in combined PWM mode.
- Bug Fixes
 - Fixed issues about channel MSnB:MSnA and ELSnB:ELSnA bit fields and CnV register change request acknowledgement. Writes to these bits are ignored when the interval between successive writes is less than the TPM clock period.

[2.0.8]

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

[2.0.7]

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

[2.0.6]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

TRDC**[2.2.2]**

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

[2.2.1]

- Bug Fixes:
 - Fix MISRA violations.

[2.2.0]

- New Features:
 - Supported SoCs that do not have all TRDC modules.

[2.1.0]

- Bug Fixes:
 - Fix MISRA violations.
 - Fixed wrong operation of domain mask in TRDC_MbcNseClearAll and TRDC_MrcDomainNseClear.

[2.0.0]

- Initial version.
-

TSTMR

[2.1.0]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

WDOG32

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.

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.

[*MIMX9352*](#)

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*](#)

1.7.2 FreeMASTER

[*freemaster*](#)

1.7.3 FreeRTOS

[*FreeRTOS*](#)

1.7.4 lwIP

[*lwip*](#)

Chapter 2

MIMX9352

2.1 eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

void EDMA_Init(*EDMA_Type* *base, const *edma_config_t* *config)

Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure. All emda enabled request will be cleared in this function.

Note: This function enables the minor loop map feature.

Parameters

- base – eDMA peripheral base address.
- config – A pointer to the configuration structure, see “edma_config_t”.

void EDMA_Deinit(*EDMA_Type* *base)

Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- base – eDMA peripheral base address.

void EDMA_InstallTCD(*EDMA_Type* *base, uint32_t channel, *edma_tcd_t* *tcd)

Push content of TCD structure into hardware TCD register.

Parameters

- base – EDMA peripheral base address.
- channel – EDMA channel number.
- tcd – Point to TCD structure.

void EDMA_GetDefaultConfig(*edma_config_t* *config)

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values.

```
config.enableContinuousLinkMode = false;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
```

Parameters

- config – A pointer to the eDMA configuration structure.

void EDMA_ResetChannel(*EDMA_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(*EDMA_Type* *base, uint32_t channel, const
 edma_transfer_config_t *config, *edma_tcd_t* *nextTcd)

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_t config;
edma_tcd_t tcd;
config.srcAddr = ..;
config.destAddr = ..;
...
EDMA_SetTransferConfig(DMA0, channel, &config, &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_SetMinorOffsetConfig(*EDMA_Type* *base, uint32_t channel, const
 edma_minor_offset_config_t *config)

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – A pointer to the minor offset configuration structure.

```
void EDMA_SetChannelPreemptionConfig(EDMA_Type *base, uint32_t channel, const  
                                     edma_channel_preemption_config_t *config)
```

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- config – A pointer to the channel preemption configuration structure.

```
void EDMA_SetChannelLink(EDMA_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_SetModulo(EDMA_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_EnableAutoStopRequest(EDMA_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(EDMA_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(EDMA_Type *base, uint32_t channel, uint32_t mask)
```

Disables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of the interrupt source to be set. Use the defined `edma_interrupt_enable_t` type.

```
void EDMA_SetMajorOffsetConfig(EDMA_Type *base, uint32_t channel, int32_t sourceOffset, int32_t destOffset)
```

Configures the eDMA channel TCD major offset feature.

Adjustment value added to the source address at the completion of the major iteration count

Parameters

- base – eDMA peripheral base address.
- channel – edma channel number.
- sourceOffset – source address offset will be applied to source address after major loop done.
- destOffset – destination address offset will be applied to source address after major loop done.

```
void EDMA_ConfigChannelSoftwareTCD(edma_tcd_t *tcd, const edma_transfer_config_t *transfer)
```

Sets TCD fields according to the user's channel transfer configuration structure, `edma_transfer_config_t`.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API `EDMA_ConfigChannelSoftwareTCDExt`

Application should be careful about the TCD pool buffer storage class,

- For the platform has cache, the software TCD should be put in non cache section
- The TCD pool buffer should have a consistent storage class.

Note: This function enables the auto stop request feature.

Parameters

- `tcd` – Pointer to the TCD structure.
- `transfer` – channel transfer configuration pointer.

void EDMA_TcdReset(*edma_tcd_t* *tcd)

Sets all fields to default values for the TCD structure.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdResetExt

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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdSetTransferConfigExt

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The TCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_t config = {
...
}
edma_tcd_t tcd __aligned(32);
edma_tcd_t nextTcd __aligned(32);
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

Parameters

- `tcd` – Pointer to the TCD structure.
- `config` – Pointer to eDMA transfer configuration structure.
- `nextTcd` – Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

```
void EDMA__TcdSetMinorOffsetConfig(edma_tcd_t *tcd, const edma_minor_offset_config_t
                                   *config)
```

Configures the eDMA TCD minor offset feature.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdSetMinorOffsetConfigExt

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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdSetChannelLinkExt

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.

```
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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdSetModuloExt

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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdEnableAutoStopRequestExt

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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdEnableInterruptsExt

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.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdDisableInterruptsExt

Parameters

- *tcd* – Point to the TCD structure.
- *mask* – The mask of interrupt source to be set. Users need to use the defined *edma_interrupt_enable_t* type.

```
void EDMA_TcdSetMajorOffsetConfig(edma_tcd_t *tcd, int32_t sourceOffset, int32_t destOffset)
```

Configures the eDMA TCD major offset feature.

@Note This API only supports EDMA4 TCD type. It can be used to support all types with extension API EDMA_TcdSetMajorOffsetConfigExt

Adjustment value added to the source address at the completion of the major iteration count

Parameters

- *tcd* – A point to the TCD structure.
- *sourceOffset* – source address offset will be applied to source address after major loop done.
- *destOffset* – destination address offset will be applied to source address after major loop done.

```
void EDMA_ConfigChannelSoftwareTCDExt(EDMA_Type *base, edma_tcd_t *tcd, const  
                                     edma_transfer_config_t *transfer)
```

Sets TCD fields according to the user's channel transfer configuration structure, *edma_transfer_config_t*.

Application should be careful about the TCD pool buffer storage class,

- For the platform has cache, the software TCD should be put in non cache section
- The TCD pool buffer should have a consistent storage class.

Note: This function enables the auto stop request feature.

Parameters

- base – eDMA peripheral base address.
- tcd – Pointer to the TCD structure.
- transfer – channel transfer configuration pointer.

void EDMA_TcdResetExt(*EDMA_Type* *base, *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

- base – eDMA peripheral base address.
- tcd – Pointer to the TCD structure.

void EDMA_TcdSetTransferConfigExt(*EDMA_Type* *base, *edma_tcd_t* *tcd, const
edma_transfer_config_t *config, *edma_tcd_t* *nextTcd)

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The TCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_t config = {  
    ...  
}  
edma_tcd_t tcd __aligned(32);  
edma_tcd_t nextTcd __aligned(32);  
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

Parameters

- base – eDMA peripheral base address.
- 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__TcdSetMinorOffsetConfigExt(EDMA_Type *base, 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

- base – eDMA peripheral base address.
- tcd – A point to the TCD structure.
- config – A pointer to the minor offset configuration structure.

```
void EDMA__TcdSetChannelLinkExt(EDMA_Type *base, 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

- base – eDMA peripheral base address.
- 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.

```
void EDMA__TcdSetModuloExt(EDMA_Type *base, 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

- base – eDMA peripheral base address.
- tcd – A pointer to the TCD structure.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

```
static inline void EDMA__TcdEnableAutoStopRequestExt(EDMA_Type *base, 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

- base – eDMA peripheral base address.
- tcd – A pointer to the TCD structure.
- enable – The command to enable (true) or disable (false).

void EDMA__TcdEnableInterruptsExt(*EDMA_Type* *base, *edma_tcd_t* *tcd, uint32_t mask)

Enables the interrupt source for the eDMA TCD.

Parameters

- base – eDMA peripheral base address.
- 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__TcdDisableInterruptsExt(*EDMA_Type* *base, *edma_tcd_t* *tcd, uint32_t mask)

Disables the interrupt source for the eDMA TCD.

Parameters

- base – eDMA peripheral base address.
- 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__TcdSetMajorOffsetConfigExt(*EDMA_Type* *base, *edma_tcd_t* *tcd, int32_t sourceOffset, int32_t destOffset)

Configures the eDMA TCD major offset feature.

Adjustment value added to the source address at the completion of the major iteration count

Parameters

- base – eDMA peripheral base address.
- tcd – A point to the TCD structure.
- sourceOffset – source address offset will be applied to source address after major loop done.
- destOffset – destination address offset will be applied to source address after major loop done.

static inline void EDMA__EnableChannelRequest(*EDMA_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(*EDMA_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(EDMA_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(EDMA_Type *base, uint32_t channel)
```

Gets the remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.

- a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTES(initially configured)
-

Parameters

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

Returns

Major loop count which has not been transferred yet for the current TCD.

```
static inline uint32_t EDMA_GetErrorStatusFlags(EDMA_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(EDMA_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(EDMA_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.

`status_t EDMA_CreateHandle(edma_handle_t *handle, EDMA_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.

Return values

- `kStatus_Success` –
- `kStatus_InvalidArgument` –

`void EDMA_InstallTCDMemory(edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)`

Installs the TCDs memory pool into the eDMA handle.

This function is called after the `EDMA_CreateHandle` to use scatter/gather feature. This function shall only be used while users need to use scatter gather mode. Scatter gather mode enables EDMA to load a new transfer control block (tcd) in hardware, and automatically reconfigure that DMA channel for a new transfer. Users need to prepare tcd memory and also configure tcds using interface `EDMA_SubmitTransfer`.

Parameters

- handle – eDMA handle pointer.
- tcdPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- tcdSize – The number of TCD slots.

`void EDMA_SetCallback(edma_handle_t *handle, edma_callback callback, void *userData)`

Installs a callback function for the eDMA transfer.

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes. This function will be called every time one tcd finished transfer.

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

`void EDMA_PrepareTransferConfig(edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)`

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE). User can check if 128 bytes support is available for specific instance by FSL_FEATURE_EDMA_INSTANCE_SUPPORT_128_BYTES_TRANSFERn.

Parameters

- config – The user configuration structure of type `edma_transfer_t`.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- srcOffset – source address offset.
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- destOffset – destination address offset.
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.

```
void EDMA__PrepareTransfer(edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth,  
                           void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest,  
                           uint32_t transferBytes, edma_transfer_type_t type)
```

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type `edma_transfer_t`.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.
- type – eDMA transfer type.

```
void EDMA__PrepareTransferTCD(edma_handle_t *handle, edma_tcd_t *tcd, void *srcAddr,  
                             uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t  
                             destWidth, int16_t destOffset, uint32_t bytesEachRequest,  
                             uint32_t transferBytes, edma_tcd_t *nextTcd)
```

Prepares the eDMA transfer content descriptor.

This function prepares the transfer content descriptor 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

- `handle` – eDMA handle pointer.
- `tcd` – Pointer to eDMA transfer content descriptor structure.
- `srcAddr` – eDMA transfer source address.
- `srcWidth` – eDMA transfer source address width(bytes).
- `srcOffset` – source address offset.
- `destAddr` – eDMA transfer destination address.
- `destWidth` – eDMA transfer destination address width(bytes).
- `destOffset` – destination address offset.
- `bytesEachRequest` – eDMA transfer bytes per channel request.
- `transferBytes` – eDMA transfer bytes to be transferred.
- `nextTcd` – eDMA transfer linked TCD address.

status_t EDMA_SubmitTransferTCD(*edma_handle_t* *handle, *edma_tcd_t* *tcd)

Submits the eDMA transfer content descriptor.

This function submits the eDMA transfer request according to the transfer content descriptor. In scatter gather mode, call this function will add a configured tcd to the circular list of tcd pool. The tcd pools is setup by call function EDMA_InstallTCDMemory before.

Typical user case:

a. submit single transfer

```
edma_tcd_t tcd;  
EDMA_PrepareTransferTCD(handle, tcd, ...)  
EDMA_SubmitTransferTCD(handle, tcd)  
EDMA_StartTransfer(handle)
```

b. submit static link transfer,

```
edma_tcd_t tcd[2];  
EDMA_PrepareTransferTCD(handle, &tcd[0], ...)  
EDMA_PrepareTransferTCD(handle, &tcd[1], ...)  
EDMA_SubmitTransferTCD(handle, &tcd[0])  
EDMA_StartTransfer(handle)
```

c. submit dynamic link transfer

```
edma_tcd_t tcdpool[2];  
EDMA_InstallTCDMemory(&g_DMA_Handle, tcdpool, 2);  
edma_tcd_t tcd;  
EDMA_PrepareTransferTCD(handle, tcd, ...)  
EDMA_SubmitTransferTCD(handle, tcd)  
EDMA_PrepareTransferTCD(handle, tcd, ...)
```

(continues on next page)

(continued from previous page)

```
EDMA_SubmitTransferTCD(handle, tcd)
EDMA_StartTransfer(handle)
```

d. submit loop transfer

```
edma_tcd_t tcd[2];
EDMA_PrepareTransferTCD(handle, &tcd[0], ..., &tcd[1])
EDMA_PrepareTransferTCD(handle, &tcd[1], ..., &tcd[0])
EDMA_SubmitTransferTCD(handle, &tcd[0])
EDMA_StartTransfer(handle)
```

Parameters

- handle – eDMA handle pointer.
- tcd – Pointer to eDMA transfer content descriptor 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.

status_t EDMA_SubmitTransfer(*edma_handle_t* *handle, const *edma_transfer_config_t* *config)
Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. In scatter gather mode, call this function will add a configured tcd to the circular list of tcd pool. The tcd pools is setup by call function EDMA_InstallTCDDMemory before.

Parameters

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

Return values

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

status_t EDMA_SubmitLoopTransfer(*edma_handle_t* *handle, *edma_transfer_config_t* *transfer, *uint32_t* transferLoopCount)

Submits the eDMA scatter gather transfer configurations.

The function is target for submit loop transfer request, the ring transfer request means that the transfer request TAIL is link to HEAD, such as, A->B->C->D->A, or A->A

To use the ring transfer feature, the application should allocate several transfer object, such as

```
edma_channel_transfer_config_t transfer[2];
EDMA_TransferSubmitLoopTransfer(psHandle, &transfer, 2U);
```

Then eDMA driver will link transfer[0] and transfer[1] to each other

Note: Application should check the return value of this function to avoid transfer request submit failed

Parameters

- `handle` – eDMA handle pointer
- `transfer` – pointer to user's eDMA channel configure structure, see `edma_channel_transfer_config_t` for detail
- `transferLoopCount` – the count of the transfer ring, if loop count is 1, that means that the one will link to itself.

Return values

- `kStatus_Success` – It means submit transfer request succeed
- `kStatus_EDMA_Busy` – channel is in busy status
- `kStatus_InvalidArgument` – Invalid Argument

`void EDMA_StartTransfer(edma_handle_t *handle)`

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- `handle` – eDMA handle pointer.

`void EDMA_StopTransfer(edma_handle_t *handle)`

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call `EDMA_StartTransfer()` again to resume the transfer.

Parameters

- `handle` – eDMA handle pointer.

`void EDMA_AbortTransfer(edma_handle_t *handle)`

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- `handle` – DMA handle pointer.

`static inline uint32_t EDMA_GetUnusedTCDNumber(edma_handle_t *handle)`

Get unused TCD slot number.

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

- `handle` – DMA handle pointer.

Returns

The unused tcd slot number.

`static inline uint32_t EDMA_GetNextTCDAddress(edma_handle_t *handle)`

Get the next tcd address.

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

- handle – DMA handle pointer.

Returns

The next TCD address.

void EDMA_HandleIRQ(*edma_handle_t* *handle)

eDMA IRQ handler for the current major loop transfer completion.

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

Note: For the case using TCD queue, when the major iteration count is exhausted, additional operations are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As sga and sga_index are calculated based on the DLAST_SGA bitfield lies in the TCD_CSR register, the sga_index in this case should be 2 (DLAST_SGA of TCD[1] stores the address of TCD[2]). Thus, the “tcdUsed” updated should be (tcdUsed - 2U) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already.).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and tcdUsed updated are identical for them. tcdUsed are both 0 in this case as no TCD to be loaded.

See the “eDMA basic data flow” in the eDMA Functional description section of the Reference Manual for further details.

Parameters

- handle – eDMA handle pointer.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.10.7.

_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_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enum _edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes

Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes

Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes

Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes

Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes

Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes

Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes

Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes

Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes

Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes

Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes

Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes

Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes

Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes

Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes

Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes

Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes

Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes

Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes

Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes

Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes

Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes

Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes

Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes

Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes

Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes

Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes

Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes

Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes

Circular buffer size is 2 G bytes.

enum _edma_channel_link_type

Channel link type.

Values:

enumerator kEDMA_LinkNone

No channel link

enumerator kEDMA_MinorLink

Channel link after each minor loop

enumerator kEDMA_MajorLink

Channel link while major loop count exhausted

_edma_channel_status_flags eDMA channel status flags.

Values:

enumerator kEDMA_DoneFlag

DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag

eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag

eDMA interrupt flag, set while an interrupt occurred of this channel

_edma_error_status_flags eDMA channel error status flags.

Values:

enumerator kEDMA_DestinationBusErrorFlag

Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag

Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag

Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag

NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag

Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag

Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag

Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag

Source address not aligned with source size

enumerator kEDMA_ErrorChannelFlag

Error channel number of the cancelled channel number

enumerator kEDMA_TransferCanceledFlag

Transfer cancelled

enumerator kEDMA_ValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

_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

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_channel_link_type* edma_channel_link_type_t

Channel link type.

typedef enum *_edma_transfer_type* edma_transfer_type_t

eDMA transfer type

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 *edma_core_tcd_t* 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 struct *_edma_transfer_config* edma_transfer_config_t

edma4 channel transfer configuration

The transfer configuration structure support full feature configuration of the transfer control descriptor.

1.To perform a simple transfer, below members should be initialized at least .srcAddr - source address .dstAddr - destination address .srcWidthOfEachTransfer - data width of source address .dstWidthOfEachTransfer - data width of destination address, normally it should be as same as srcWidthOfEachTransfer .bytesEachRequest - bytes to be transferred in each DMA request .totalBytes - total bytes to be transferred .srcOffsetOfEachTransfer - offset value in bytes unit to be applied to source address as each source read is completed .dstOffsetOfEachTransfer - offset value in bytes unit to be applied to destination address as each destination write is completed enablchannelRequest - channel request can be enabled together with transfer configure submission

2.The transfer configuration structure also support advance feature: Programmable source/destination address range(MODULO) Programmable minor loop offset Programmable major loop offset Programmable channel chain feature Programmable channel transfer control descriptor link feature

Note: User should pay attention to the transfer size alignment limitation

- a. the bytesEachRequest should align with the srcWidthOfEachTransfer and the dstWidthOfEachTransfer that is to say bytesEachRequest % srcWidthOfEachTransfer should be 0

- b. the `srcOffsetOfEachTransfer` and `dstOffsetOfEachTransfer` must be aligne with transfer width
 - c. the `totalBytes` should align with the `bytesEachRequest`
 - d. the `srcAddr` should align with the `srcWidthOfEachTransfer`
 - e. the `dstAddr` should align with the `dstWidthOfEachTransfer`
 - f. the `srcAddr` should align with `srcAddrModulo` if modulo feature is enabled
 - g. the `dstAddr` should align with `dstAddrModulo` if modulo feature is enabled If anyone of above condition can not be satisfied, the edma4 interfaces will generate assert error.
-

```
typedef struct _edma_config edma_config_t
```

eDMA global configuration structure.

```
typedef void (*edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone,
uint32_t tcds)
```

Define callback function for eDMA.

This callback function is called in the EDMA interrupt handle. In normal mode, run into callback function means the transfer users need is done. In scatter gather mode, run into callback function means a transfer control block (tcd) is finished. Not all transfer finished, users can get the finished tcd numbers using interface `EDMA_GetUnusedTCDNumber`.

Param handle

EDMA handle pointer, users shall not touch the values inside.

Param userData

The callback user parameter pointer. Users can use this parameter to involve things users need to change in EDMA callback function.

Param transferDone

If the current loaded transfer done. In normal mode it means if all transfer done. In scatter gather mode, this parameter shows is the current transfer block in EDMA register is done. As the load of core is different, it will be different if the new tcd loaded into EDMA registers while this callback called. If true, it always means new tcd still not loaded into registers, while false means new tcd already loaded into registers.

Param tcds

How many tcds are done from the last callback. This parameter only used in scatter gather mode. It tells user how many tcds are finished between the last callback and this.

```
typedef struct _edma_handle edma_handle_t
```

eDMA transfer handle structure

```
FSL_EDMA_DRIVER_EDMA4
```

eDMA driver name

```
EDMA_ALLOCATE_TCD(name, number)
```

Macro used for allocate edma TCD.

```
DMA_DCHPRI_INDEX(channel)
```

Compute the offset unit from DCHPRI3.

```
struct _edma_channel_Preemption_config
```

`#include <fsl_edma.h>` eDMA channel priority configuration

Public Members

bool enableChannelPreemption

If true: a channel can be suspended by other channel with higher priority

bool enablePreemptAbility

If true: a channel can suspend other channel with low priority

uint8_t channelPriority

Channel priority

struct _edma_minor_offset_config

#include <fsl_edma.h> eDMA minor offset configuration

Public Members

bool enableSrcMinorOffset

Enable(true) or Disable(false) source minor loop offset.

bool enableDestMinorOffset

Enable(true) or Disable(false) destination minor loop offset.

uint32_t minorOffset

Offset for a minor loop mapping.

struct _edma_transfer_config

#include <fsl_edma.h> edma4 channel transfer configuration

The transfer configuration structure support full feature configuration of the transfer control descriptor.

1.To perform a simple transfer, below members should be initialized at least .srcAddr - source address .dstAddr - destination address .srcWidthOfEachTransfer - data width of source address .dstWidthOfEachTransfer - data width of destination address, normally it should be as same as srcWidthOfEachTransfer .bytesEachRequest - bytes to be transferred in each DMA request .totalBytes - total bytes to be transferred .srcOffsetOfEachTransfer - offset value in bytes unit to be applied to source address as each source read is completed .dstOffsetOfEachTransfer - offset value in bytes unit to be applied to destination address as each destination write is completed enablchannelRequest - channel request can be enabled together with transfer configure submission

2.The transfer configuration structure also support advance feature: Programmable source/destination address range(MODULO) Programmable minor loop offset Programmable major loop offset Programmable channel chain feature Programmable channel transfer control descriptor link feature

Note: User should pay attention to the transfer size alignment limitation

- the bytesEachRequest should align with the srcWidthOfEachTransfer and the dstWidthOfEachTransfer that is to say bytesEachRequest % srcWidthOfEachTransfer should be 0
- the srcOffsetOfEachTransfer and dstOffsetOfEachTransfer must be aligne with transfer width
- the totalBytes should align with the bytesEachRequest
- the srcAddr should align with the srcWidthOfEachTransfer
- the dstAddr should align with the dstWidthOfEachTransfer
- the srcAddr should align with srcAddrModulo if modulo feature is enabled

- g. the dstAddr should align with dstAddrModulo if modulo feature is enabled If anyone of above condition can not be satisfied, the edma4 interfaces will generate assert error.
-

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 value in byte unit applied to the current source address to form the next-state value as each source read is completed

int16_t destOffset

Sign-extended offset value in byte unit applied to the current destination address to form the next-state value as each destination write is completed.

uint32_t minorLoopBytes

bytes in each minor loop or each request range: 1 - ($2^{30} - 1$) when minor loop mapping is enabled range: 1 - ($2^{10} - 1$) when minor loop mapping is enabled and source or dest minor loop offset is enabled range: 1 - ($2^{32} - 1$) when minor loop mapping is disabled

uint32_t majorLoopCounts

minor loop counts in each major loop, should be 1 at least for each transfer range: (0 - ($2^{15} - 1$)) when minor loop channel link is disabled range: (0 - ($2^9 - 1$)) when minor loop channel link is enabled total bytes in a transfer = minorLoopCountsEachMajorLoop * bytesEachMinorLoop

uint16_t enabledInterruptMask

channel interrupt to enable, can be OR'ed value of _edma_interrupt_enable

edma_modulo_t srcAddrModulo

source circular data queue range

int32_t srcMajorLoopOffset

source major loop offset

edma_modulo_t dstAddrModulo

destination circular data queue range

int32_t dstMajorLoopOffset

destination major loop offset

bool enableSrcMinorLoopOffset

enable source minor loop offset

bool enableDstMinorLoopOffset

enable dest minor loop offset

int32_t minorLoopOffset

burst offset, the offset will be applied after minor loop update

`bool enableChannelMajorLoopLink`
channel link when major loop complete

`uint32_t majorLoopLinkChannel`
major loop link channel number

`bool enableChannelMinorLoopLink`
channel link when minor loop complete

`uint32_t minorLoopLinkChannel`
minor loop link channel number

`edma_tcd_t *linkTCD`
pointer to the link transfer control descriptor

`struct __edma_config`
#include <fsl_edma.h> eDMA global configuration structure.

Public Members

`bool enableGlobalChannelLink`
Enable(true) channel linking is available and controlled by each channel's link settings.

`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 enableDebugMode`
Enable(true) eDMA4 debug mode. When in debug mode, the eDMA4 stalls the start of a new channel. Executing channels are allowed to complete.

`bool enableRoundRobinArbitration`
Enable(true) channel linking is available and controlled by each channel's link settings.

`struct __edma_handle`
#include <fsl_edma.h> eDMA transfer handle structure

Public Members

`edma_callback` callback
Callback function for major count exhausted.

`void *userData`
Callback function parameter.

`EDMA_Type *base`
eDMA peripheral base address

`EDMA_TCDType *tcdBase`
eDMA peripheral tcd base address.

`edma_tcd_t *tcdPool`
Pointer to memory stored TCDs.

`uint32_t channel`
eDMA channel number.

`volatile int8_t header`
The first TCD index. Should point to the next TCD to be loaded into the eDMA engine.

volatile int8_t tail

The last TCD index. Should point to the next TCD to be stored into the memory pool.

volatile int8_t tcdUsed

The number of used TCD slots. Should reflect the number of TCDs can be used/loaded in the memory.

volatile int8_t tcdSize

The total number of TCD slots in the queue.

2.2 eDMA core Driver

enum _edma_tcd_type

eDMA tcd flag type

Values:

enumerator kEDMA_EDMA4Flag

Data access for eDMA4 transfers.

enumerator kEDMA_EDMA5Flag

Instruction access for eDMA4 transfers.

typedef struct _edma_core_mp edma_core_mp_t

edma core channel struture definition

typedef struct _edma_core_channel edma_core_channel_t

edma core channel struture definition

typedef enum _edma_tcd_type edma_tcd_type_t

eDMA tcd flag type

typedef struct _edma5_core_tcd edma5_core_tcd_t

edma5 core TCD struture definition

typedef struct _edma4_core_tcd edma4_core_tcd_t

edma4 core TCD struture definition

typedef struct _edma_core_tcd edma_core_tcd_t

edma core TCD struture definition

typedef edma_core_channel_t EDMA_ChannelType

EDMA typedef.

typedef edma_core_tcd_t EDMA_TCDType

typedef void EDMA_Type

DMA_CORE_MP_CSR_EDBG_MASK

DMA_CORE_MP_CSR_ERCA_MASK

DMA_CORE_MP_CSR_HAE_MASK

DMA_CORE_MP_CSR_HALT_MASK

DMA_CORE_MP_CSR_GCLC_MASK

DMA_CORE_MP_CSR_GMRC_MASK

DMA_CORE_MP_CSR_EDBG(x)

DMA_CORE_MP_CSR_ERCA(x)
DMA_CORE_MP_CSR_HAE(x)
DMA_CORE_MP_CSR_HALT(x)
DMA_CORE_MP_CSR_GCLC(x)
DMA_CORE_MP_CSR_GMRC(x)
DMA_CSR_INTMAJOR_MASK
DMA_CSR_INTHALF_MASK
DMA_CSR_DREQ_MASK
DMA_CSR_ESG_MASK
DMA_CSR_BWC_MASK
DMA_CSR_BWC(x)
DMA_CSR_START_MASK
DMA_CITER_ELINKNO_CITER_MASK
DMA_BITER_ELINKNO_BITER_MASK
DMA_CITER_ELINKNO_CITER_SHIFT
DMA_CITER_ELINKYES_CITER_MASK
DMA_CITER_ELINKYES_CITER_SHIFT
DMA_ATTR_SMOD_MASK
DMA_ATTR_DMOD_MASK
DMA_CITER_ELINKNO_ELINK_MASK
DMA_CSR_MAJORELINK_MASK
DMA_BITER_ELINKYES_ELINK_MASK
DMA_CITER_ELINKYES_ELINK_MASK
DMA_CSR_MAJORLINKCH_MASK
DMA_BITER_ELINKYES_LINKCH_MASK
DMA_CITER_ELINKYES_LINKCH_MASK
DMA_NBYTES_MLOFFYES_MLOFF_MASK
DMA_NBYTES_MLOFFYES_DMLOE_MASK
DMA_NBYTES_MLOFFYES_SMLOE_MASK
DMA_NBYTES_MLOFFNO_NBYTES_MASK
DMA_ATTR_DMOD(x)
DMA_ATTR_SMOD(x)
DMA_BITER_ELINKYES_LINKCH(x)

DMA_CITER_ELINKYES_LINKCH(x)
DMA_NBYTES_MLOFFYES_MLOFF(x)
DMA_NBYTES_MLOFFYES_DMLOE(x)
DMA_NBYTES_MLOFFYES_SMLOE(x)
DMA_NBYTES_MLOFFNO_NBYTES(x)
DMA_NBYTES_MLOFFYES_NBYTES(x)
DMA_ATTR_DSIZE(x)
DMA_ATTR_SSIZE(x)
DMA_CSR_DREQ(x)
DMA_CSR_MAJORLINKCH(x)
DMA_CH_MATTR_WCACHE(x)
DMA_CH_MATTR_RCACHE(x)
DMA_CH_CSR_SIGNEXT_MASK
DMA_CH_CSR_SIGNEXT_SHIFT
DMA_CH_CSR_SWAP_MASK
DMA_CH_CSR_SWAP_SHIFT
DMA_CH_SBR_INSTR_MASK
DMA_CH_SBR_INSTR_SHIFT
DMA_CH_SBR_EMI_MASK
DMA_CH_SBR_EMI_SHIFT
DMA_CH_MUX_SOURCE(x)
DMA_ERR_DBE_FLAG
DMA error flag.
DMA_ERR_SBE_FLAG
DMA_ERR_SGE_FLAG
DMA_ERR_NCE_FLAG
DMA_ERR_DOE_FLAG
DMA_ERR_DAE_FLAG
DMA_ERR_SOE_FLAG
DMA_ERR_SAE_FLAG
DMA_ERR_ERRCHAN_FLAG
DMA_ERR_ECX_FLAG
DMA_ERR_FLAG

DMA_CLEAR_DONE_STATUS(base, channel)
get/clear DONE bit

DMA_GET_DONE_STATUS(base, channel)

DMA_ENABLE_ERROR_INT(base, channel)
enable/disable error interrupt

DMA_DISABLE_ERROR_INT(base, channel)

DMA_CLEAR_ERROR_STATUS(base, channel)
get/clear error status

DMA_GET_ERROR_STATUS(base, channel)

DMA_CLEAR_INT_STATUS(base, channel)
get/clear INT status

DMA_GET_INT_STATUS(base, channel)

DMA_ENABLE_MAJOR_INT(base, channel)
enable/disable MAJOR/HALF INT

DMA_ENABLE_HALF_INT(base, channel)

DMA_DISABLE_MAJOR_INT(base, channel)

DMA_DISABLE_HALF_INT(base, channel)

EDMA_TCD_ALIGN_SIZE
EDMA tcd align size.

EDMA_CORE_BASE(base)
EDMA base address convert macro.

EDMA_MP_BASE(base)

EDMA_CHANNEL_BASE(base, channel)

EDMA_TCD_BASE(base, channel)

EDMA_TCD_TYPE(x)
EDMA TCD type macro.

EDMA_TCD_SADDR(tcd, flag)
EDMA TCD address convert macro.

EDMA_TCD_SOFF(tcd, flag)

EDMA_TCD_ATTR(tcd, flag)

EDMA_TCD_NBYTES(tcd, flag)

EDMA_TCD_SLAST(tcd, flag)

EDMA_TCD_DADDR(tcd, flag)

EDMA_TCD_DOFF(tcd, flag)

EDMA_TCD_CITER(tcd, flag)

EDMA_TCD_DLAST_SGA(tcd, flag)

EDMA_TCD_CSR(tcd, flag)

EDMA_TCD_BITER(tcd, flag)

struct __edma_core_mp

#include <fsl_edma_core.h> edma core channel struture definition

Public Members

__IO uint32_t MP_CSR

Channel Control and Status, array offset: 0x10000, array step: 0x10000

__IO uint32_t MP_ES

Channel Error Status, array offset: 0x10004, array step: 0x10000

struct __edma_core_channel

#include <fsl_edma_core.h> edma core channel struture definition

Public Members

__IO uint32_t CH_CSR

Channel Control and Status, array offset: 0x10000, array step: 0x10000

__IO uint32_t CH_ES

Channel Error Status, array offset: 0x10004, array step: 0x10000

__IO uint32_t CH_INT

Channel Interrupt Status, array offset: 0x10008, array step: 0x10000

__IO uint32_t CH_SBR

Channel System Bus, array offset: 0x1000C, array step: 0x10000

__IO uint32_t CH_PRI

Channel Priority, array offset: 0x10010, array step: 0x10000

struct __edma5_core_tcd

#include <fsl_edma_core.h> edma5 core TCD struture definition

Public Members

__IO uint32_t SADDR

SADDR register, used to save source address

__IO uint32_t SADDR_HIGH

SADDR HIGH 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 SLAST_SDA_HIGH

SLAST SDA HIGH register

__IO uint32_t DADDR
DADDR register, used for destination address

__IO uint32_t DADDR_HIGH
DADDR HIGH register, used for destination address

__IO uint32_t DLAST_SGA
DLASTSGA register, next tcd address used in scatter-gather mode

__IO uint32_t DLAST_SGA_HIGH
DLASTSGA HIGH register, next tcd address used in scatter-gather mode

__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 uint16_t CSR
CSR register, for TCD control status

__IO uint16_t BITER
BITER register, begin minor loop count.

uint8_t RESERVED[16]
Aligned 64 bytes

struct _edma4_core_tcd
#include <fsl_edma_core.h> edma4 core TCD struture definition

Public Members

__IO uint32_t SADDR
SADDR register, used to save source address

__IO uint16_t SOFF
SOFF register, save offset bytes every transfer

__IO uint16_t ATTR
ATTR register, source/destination transfer size and modulo

__IO uint32_t NBYTES
Nbytes register, minor loop length in bytes

__IO uint32_t SLAST
SLAST register

__IO uint32_t DADDR
DADDR register, used for destination address

__IO uint16_t DOFF
DOFF register, used for destination offset

__IO uint16_t CITER
CITER register, current minor loop numbers, for unfinished minor loop.

__IO uint32_t DLAST_SGA
DLASTSGA register, next tcd address used in scatter-gather mode

__IO uint16_t CSR
CSR register, for TCD control status

```
__IO uint16_t BITER
    BITER register, begin minor loop count.
struct _edma_core_tcd
    #include <fsl_edma_core.h> edma core TCD struture definition
union MP_REGS
```

Public Members

```
struct _edma_core_mp EDMA5_REG
struct EDMA5_REG
```

Public Members

```
__IO uint32_t MP_INT_LOW
    Channel Control and Status, array offset: 0x10008, array step: 0x10000
__I uint32_t MP_INT_HIGH
    Channel Control and Status, array offset: 0x1000C, array step: 0x10000
__I uint32_t MP_HRS_LOW
    Channel Control and Status, array offset: 0x10010, array step: 0x10000
__I uint32_t MP_HRS_HIGH
    Channel Control and Status, array offset: 0x10014, array step: 0x10000
__IO uint32_t MP_STOPCH
    Channel Control and Status, array offset: 0x10020, array step: 0x10000
__I uint32_t MP_SSR_LOW
    Channel Control and Status, array offset: 0x10030, array step: 0x10000
__I uint32_t MP_SSR_HIGH
    Channel Control and Status, array offset: 0x10034, array step: 0x10000
__IO uint32_t CH_GRPRI [64]
    Channel Control and Status, array offset: 0x10100, array step: 0x10000
__IO uint32_t CH_MUX [64]
    Channel Control and Status, array offset: 0x10200, array step: 0x10000
__IO uint32_t CH_PROT [64]
    Channel Control and Status, array offset: 0x10400, array step: 0x10000
union CH_REGS
```

Public Members

```
struct _edma_core_channel EDMA5_REG
struct _edma_core_channel EDMA4_REG
struct EDMA5_REG
```

Public Members

__IO uint32_t CH_MATTR
Memory Attributes Register, array offset: 0x10018, array step: 0x8000
struct EDMA4_REG

Public Members

__IO uint32_t CH_MUX
Channel Multiplexor Configuration, array offset: 0x10014, array step: 0x10000
__IO uint16_t CH_MATTR
Memory Attributes Register, array offset: 0x10018, array step: 0x8000
union TCD_REGS

Public Members

edma4_core_tcd_t edma4_tcd

2.3 eDMA soc Driver

FSL_EDMA_SOC_DRIVER_VERSION
Driver version 2.0.0.
FSL_EDMA_SOC_IP_DMA3
DMA IP version.
FSL_EDMA_SOC_IP_DMA4
EDMA_BASE_PTRS
DMA base table.
EDMA_CHN_IRQS
FSL_FEATURE_EDMA_HAS_CHANNEL_MUX
Verify dma base and request source
EDMA_CHANNEL_HAS_REQUEST_SOURCE(base, source)
EDMA_CHANNEL_OFFSET
EDMA base address convert macro.
EDMA_CHANNEL_ARRAY_STEP(base)

2.4 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_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_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 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.

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.
    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_QUEUEn(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);
    }
}
```

(continues on next page)

(continued from previous page)

```
    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.

`FSL_FEATURE_ENET_QUEUE`

Defines the queue number.

`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_SOFTOWNER1_MASK

Software owner one mask.

ENET_BUFFDESCRIPTOR_TX_WRAP_MASK

Wrap buffer descriptor mask.

ENET_BUFFDESCRIPTOR_TX_SOFTOWNER2_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.

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.

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_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).

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_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_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_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_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.

```
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_BUFDESCRITOR_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 txConsumIdx
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_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_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.

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_tx_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 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.

`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 txConsumIdx

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

`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.

2.5 EQOS-TSN: Ethernet QoS with TSN Driver

2.6 Enet_qos_qos

void ENET_QOS_GetDefaultConfig(*enet_qos_config_t* *config)

Gets the ENET default configuration structure.

The purpose of this API is to get the default ENET configure structure for ENET_QOS_Init(). User may use the initialized structure unchanged in ENET_QOS_Init(), or modify some fields of the structure before calling ENET_QOS_Init(). Example:

```
enet_qos_config_t config;  
ENET_QOS_GetDefaultConfig(&config);
```

Parameters

- config – The ENET mac controller configuration structure pointer.

status_t ENET_QOS_Up(ENET_QOS_Type *base, const *enet_qos_config_t* *config, uint8_t *macAddr, uint8_t macCount, uint32_t refclkSrc_Hz)

Initializes the ENET module.

This function initializes it with the ENET basic configuration.

Parameters

- base – ENET peripheral base address.
- config – ENET mac configuration structure pointer. The “enet_qos_config_t” type mac configuration return from ENET_QOS_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
- macAddr – Pointer to ENET mac address array of Ethernet device. This MAC address should be provided.
- macCount – Count of macAddr in the ENET mac address array
- refclkSrc_Hz – ENET input reference clock.

status_t ENET_QOS_Init(ENET_QOS_Type *base, const *enet_qos_config_t* *config, uint8_t *macAddr, uint8_t macCount, uint32_t refclkSrc_Hz)

Initializes the ENET module.

This function ungates the module clock and initializes it with the ENET basic configuration.

Parameters

- base – ENET peripheral base address.

- `config` – ENET mac configuration structure pointer. The “enet_qos_config_t” type mac configuration return from `ENET_QOS_GetDefaultConfig` can be used directly. It is also possible to verify the Mac configuration using other methods.
- `macAddr` – Pointer to ENET mac address array of Ethernet device. This MAC address should be provided.
- `macCount` – Count of `macAddr` in the ENET mac address array
- `refclkSrc_Hz` – ENET input reference clock.

`void ENET_QOS_Down(ENET_QOS_Type *base)`

Stops the ENET module.

This function disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`void ENET_QOS_Deinit(ENET_QOS_Type *base)`

Deinitializes the ENET module.

This function gates the module clock and disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`uint32_t ENET_QOS_GetInstance(ENET_QOS_Type *base)`

Get the ENET instance from peripheral base address.

Parameters

- `base` – ENET peripheral base address.

Returns

ENET instance.

`status_t ENET_QOS_DescriptorInit(ENET_QOS_Type *base, enet_qos_config_t *config, enet_qos_buffer_config_t *bufferConfig)`

Initialize for all ENET descriptors.

Note: This function is do all tx/rx descriptors initialization. Because this API read all interrupt registers first and then set the interrupt flag for all descriptors, if the interrupt register is set. so the descriptor initialization should be called after `ENET_QOS_Init()`, `ENET_QOS_EnableInterrupts()` and `ENET_QOS_CreateHandle()`(if transactional APIs are used).

Parameters

- `base` – ENET peripheral base address.
- `config` – The configuration for ENET.
- `bufferConfig` – All buffers configuration.

`status_t ENET_QOS_RxBufferAllocAll(ENET_QOS_Type *base, enet_qos_handle_t *handle)`

Allocates Rx buffers for all BDs. It's used for zero copy Rx. In zero copy Rx case, Rx buffers are dynamic. This function will populate initial buffers in all BDs for receiving. Then `ENET_QOS_GetRxFrame()` is used to get Rx frame with zero copy, it will allocate new buffer to replace the buffer in BD taken by application application should free those buffers after they're used.

Note: This function should be called after ENET_QOS_CreateHandler() and buffer allocating callback function should be ready.

Parameters

- base – ENET_QOS peripheral base address.
- handle – The ENET_QOS handler structure. This is the same handler pointer used in the ENET_QOS_Init.

void ENET_QOS_RxBufferFreeAll(ENET_QOS_Type *base, *enet_qos_handle_t* *handle)

Frees Rx buffers in all BDs. It's used for zero copy Rx. In zero copy Rx case, Rx buffers are dynamic. This function will free left buffers in all BDs.

Parameters

- base – ENET_QOS peripheral base address.
- handle – The ENET_QOS handler structure. This is the same handler pointer used in the ENET_QOS_Init.

void ENET_QOS_StartRxTx(ENET_QOS_Type *base, uint8_t txRingNum, uint8_t rxRingNum)

Starts the ENET rx/tx. This function enable the tx/rx and starts the rx/tx DMA. This shall be set after ENET initialization and before starting to receive the data.

Note: This must be called after all the ENET initialization. And should be called when the ENET receive/transmit is required.

Parameters

- base – ENET peripheral base address.
- rxRingNum – The number of the used rx rings. It shall not be larger than the ENET_QOS_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.
- txRingNum – The number of the used tx rings. It shall not be larger than the ENET_QOS_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.

status_t ENET_QOS_SetMII(ENET_QOS_Type *base, *enet_qos_mii_speed_t* speed, *enet_qos_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.

Returns

kStatus_Success The ENET MII speed and duplex has been set successfully.

Returns

kStatus_InvalidArgument Could not set the desired ENET MII speed and duplex combination.

```
void ENET_QOS_SetSMI(ENET_QOS_Type *base, uint32_t csrClock_Hz)
```

Sets the ENET SMI(serial management interface)- MII management interface.

Parameters

- base – ENET peripheral base address.
- csrClock_Hz – CSR clock frequency in HZ

```
static inline bool ENET_QOS_IsSMIBusy(ENET_QOS_Type *base)
```

Checks if the SMI is busy.

Parameters

- base – ENET peripheral base address.

Returns

The status of MII Busy status.

```
static inline uint16_t ENET_QOS_ReadSMIData(ENET_QOS_Type *base)
```

Reads data from the PHY register through SMI interface.

Parameters

- base – ENET peripheral base address.

Returns

The data read from PHY

```
void ENET_QOS_StartSMIWrite(ENET_QOS_Type *base, uint8_t phyAddr, uint8_t regAddr,  
uint16_t data)
```

Sends the MDIO IEEE802.3 Clause 22 format write command. After send command, user needs to check whether the transmission is over with ENET_QOS_IsSMIBusy().

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address.
- regAddr – The PHY register address.
- data – The data written to PHY.

```
void ENET_QOS_StartSMIRead(ENET_QOS_Type *base, uint8_t phyAddr, uint8_t regAddr)
```

Sends the MDIO IEEE802.3 Clause 22 format read command. After send command, user needs to check whether the transmission is over with ENET_QOS_IsSMIBusy().

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address.
- regAddr – The PHY register address.

```
void ENET_QOS_StartExtC45SMIWrite(ENET_QOS_Type *base, uint8_t portAddr, uint8_t  
devAddr, uint16_t regAddr, uint16_t data)
```

Sends the MDIO IEEE802.3 Clause 45 format write command. After send command, user needs to check whether the transmission is over with ENET_QOS_IsSMIBusy().

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.

```
void ENET_QOS_StartExtC45SMIRRead(ENET_QOS_Type *base, uint8_t portAddr, uint8_t  
                                devAddr, uint16_t regAddr)
```

Sends the MDIO IEEE802.3 Clause 45 format read command. After send command, user needs to check whether the transmission is over with ENET_QOS_IsSMIBusy().

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.

```
status_t ENET_QOS_MDIOWrite(ENET_QOS_Type *base, uint8_t phyAddr, uint8_t regAddr,  
                             uint16_t data)
```

MDIO write with IEEE802.3 MDIO Clause 22 format.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address.
- regAddr – The PHY register.
- data – The data written to PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
status_t ENET_QOS_MDIORead(ENET_QOS_Type *base, uint8_t phyAddr, uint8_t regAddr,  
                            uint16_t *pData)
```

MDIO read with IEEE802.3 MDIO Clause 22 format.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address.
- regAddr – The PHY register.
- pData – The data read from PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
status_t ENET_QOS_MDIOWC45Write(ENET_QOS_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_QOS_MDIOC45Read(ENET_QOS_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_QOS_SetMacAddr(ENET_QOS_Type *base, uint8_t *macAddr, uint8_t index)

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.
- index – Configure macAddr to MAC_ADDRESS[index] register.

void ENET_QOS_GetMacAddr(ENET_QOS_Type *base, uint8_t *macAddr, uint8_t index)

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.
- index – Get macAddr from MAC_ADDRESS[index] register.

void ENET_QOS_AddMulticastGroup(ENET_QOS_Type *base, uint8_t *address)

Adds the ENET_QOS device to a multicast group.

Parameters

- base – ENET_QOS peripheral base address.
- address – The six-byte multicast group address which is provided by application.

void ENET_QOS_LeaveMulticastGroup(ENET_QOS_Type *base, uint8_t *address)

Moves the ENET_QOS device from a multicast group.

Parameters

- base – ENET_QOS peripheral base address.

- address – The six-byte multicast group address which is provided by application.

static inline void ENET_QOS_AcceptAllMulticast(ENET_QOS_Type *base)

Enable ENET device to accept all multicast frames.

Parameters

- base – ENET peripheral base address.

static inline void ENET_QOS_RejectAllMulticast(ENET_QOS_Type *base)

ENET device reject to accept all multicast frames.

Parameters

- base – ENET peripheral base address.

void ENET_QOS_EnterPowerDown(ENET_QOS_Type *base, uint32_t *wakeFilter)

Set the MAC to enter into power down mode. the remote power wake up frame and magic frame can wake up the ENET from the power down mode.

Parameters

- base – ENET peripheral base address.
- wakeFilter – The wakeFilter provided to configure the wake up frame filter. Set the wakeFilter to NULL is not required. But if you have the filter requirement, please make sure the wakeFilter pointer shall be eight continuous 32-bits configuration.

static inline void ENET_QOS_ExitPowerDown(ENET_QOS_Type *base)

Set the MAC to exit power down mode. Exit from the power down mode and recover to normal work mode.

Parameters

- base – ENET peripheral base address.

status_t ENET_QOS_EnableRxParser(ENET_QOS_Type *base, bool enable)

Enable/Disable Rx parser, please notice that for enable/disable Rx Parser, should better disable Receive first.

Parameters

- base – ENET_QOS peripheral base address.
- enable – Enable/Disable Rx parser function

Return values

- kStatus_Success – Configure rx parser success.
- kStatus_ENET_QOS_Timeout – Poll status flag timeout.

void ENET_QOS_EnableInterrupts(ENET_QOS_Type *base, uint32_t mask)

Enables the ENET DMA and MAC interrupts.

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_qos_dma_interrupt_enable_t and enet_qos_mac_interrupt_enable_t. For example, to enable the dma and mac interrupt, do the following.

```
ENET_QOS_EnableInterrupts(ENET, kENET_QOS_DmaRx | kENET_QOS_DmaTx | kENET_QOS_MacPmt);
```

Parameters

- base – ENET peripheral base address.

- mask – ENET interrupts to enable. This is a logical OR of both enumeration :: enet_qos_dma_interrupt_enable_t and enet_qos_mac_interrupt_enable_t.

void ENET_QOS_DisableInterrupts(ENET_QOS_Type *base, uint32_t mask)

Disables the ENET DMA and MAC interrupts.

This function disables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_qos_dma_interrupt_enable_t and enet_qos_mac_interrupt_enable_t. For example, to disable the dma and mac interrupt, do the following.

```
ENET_QOS_DisableInterrupts(ENET, kENET_QOS_DmaRx | kENET_QOS_DmaTx | kENET_QOS_MacPmt);
```

Parameters

- base – ENET peripheral base address.
- mask – ENET interrupts to disables. This is a logical OR of both enumeration :: enet_qos_dma_interrupt_enable_t and enet_qos_mac_interrupt_enable_t.

static inline uint32_t ENET_QOS_GetDmaInterruptStatus(ENET_QOS_Type *base, uint8_t channel)

Gets the ENET DMA interrupt status flag.

Parameters

- base – ENET peripheral base address.
- channel – The DMA Channel. Shall not be larger than ENET_QOS_RING_NUM_MAX.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: enet_qos_dma_interrupt_enable_t.

static inline void ENET_QOS_ClearDmaInterruptStatus(ENET_QOS_Type *base, uint8_t channel, uint32_t mask)

Clear the ENET DMA interrupt status flag.

Parameters

- base – ENET peripheral base address.
- channel – The DMA Channel. Shall not be larger than ENET_QOS_RING_NUM_MAX.
- mask – The interrupt status to be cleared. This is the logical OR of members of the enumeration :: enet_qos_dma_interrupt_enable_t.

static inline uint32_t ENET_QOS_GetMacInterruptStatus(ENET_QOS_Type *base)

Gets the ENET MAC interrupt status flag.

Parameters

- base – ENET peripheral base address.

Returns

The event status of the interrupt source. Use the enum in enet_qos_mac_interrupt_enable_t and right shift ENET_QOS_MACINT_ENUM_OFFSET to mask the returned value to get the exact interrupt status.

```
void ENET_QOS_ClearMacInterruptStatus(ENET_QOS_Type *base, uint32_t mask)
```

Clears the ENET mac 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_qos_mac_interrupt_enable_t`. For example, to clear the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_QOS_ClearMacInterruptStatus(ENET, kENET_QOS_MacPmt);
```

Parameters

- `base` – ENET peripheral base address.
- `mask` – ENET interrupt source to be cleared. This is the logical OR of members of the enumeration :: `enet_qos_mac_interrupt_enable_t`.

```
static inline bool ENET_QOS_IsTxDescriptorDmaOwn(enet_qos_tx_bd_struct_t *txDesc)
```

Get the tx descriptor DMA Own flag.

Parameters

- `txDesc` – The given tx descriptor.

Return values

True – the dma own tx descriptor, false application own tx descriptor.

```
void ENET_QOS_SetupTxDescriptor(enet_qos_tx_bd_struct_t *txDesc, void *buffer1, uint32_t bytes1, void *buffer2, uint32_t bytes2, uint32_t framelen, bool intEnable, bool tsEnable, enet_qos_desc_flag flag, uint8_t slotNum)
```

Setup a given tx descriptor. This function is a low level functional API to setup or prepare a given tx descriptor.

Note: This must be called after all the ENET initialization. And should be called when the ENET receive/transmit is required. Transmit buffers are ‘zero-copy’ buffers, so the buffer must remain in memory until the packet has been fully transmitted. The buffers should be free or queued in the transmit interrupt irq handler.

Parameters

- `txDesc` – The given tx descriptor.
- `buffer1` – The first buffer address in the descriptor.
- `bytes1` – The bytes in the fist buffer.
- `buffer2` – The second buffer address in the descriptor.
- `bytes2` – The bytes in the second buffer.
- `framelen` – The length of the frame to be transmitted.
- `intEnable` – Interrupt enable flag.
- `tsEnable` – The timestamp enable.
- `flag` – The flag of this tx descriptor, `enet_qos_desc_flag` .
- `slotNum` – The slot num used for AV only.

```
static inline void ENET_QOS_UpdateTxDescriptorTail(ENET_QOS_Type *base, uint8_t channel, uint32_t txDescTailAddrAlign)
```

Update the tx descriptor tail pointer. This function is a low level functional API to update the the tx descriptor tail. This is called after you setup a new tx descriptor to update the tail pointer to make the new descriptor accessible by DMA.

Parameters

- base – ENET peripheral base address.
- channel – The tx DMA channel.
- txDescTailAddrAlign – The new tx tail pointer address.

```
static inline void ENET_QOS_UpdateRxDescriptorTail(ENET_QOS_Type *base, uint8_t channel,  
                                                  uint32_t rxDescTailAddrAlign)
```

Update the rx descriptor tail pointer. This function is a low level functional API to update the the rx descriptor tail. This is called after you setup a new rx descriptor to update the tail pointer to make the new descriptor accessible by DMA and to anouse the rx poll command for DMA.

Parameters

- base – ENET peripheral base address.
- channel – The rx DMA channel.
- rxDescTailAddrAlign – The new rx tail pointer address.

```
static inline uint32_t ENET_QOS_GetRxDescriptor(enet_qos_rx_bd_struct_t *rxDesc)
```

Gets the context in the ENET rx descriptor. This function is a low level functional API to get the the status flag from a given rx descriptor.

Note: This must be called after all the ENET initialization. And should be called when the ENET receive/transmit is required.

Parameters

- rxDesc – The given rx descriptor.

Return values

The – RDES3 regions for write-back format rx buffer descriptor.

```
void ENET_QOS_UpdateRxDescriptor(enet_qos_rx_bd_struct_t *rxDesc, void *buffer1, void  
                                *buffer2, bool intEnable, bool doubleBuffEnable)
```

Updates the buffers and the own status for a given rx descriptor. This function is a low level functional API to Updates the buffers and the own status for a given rx descriptor.

Note: This must be called after all the ENET initialization. And should be called when the ENET receive/transmit is required.

Parameters

- rxDesc – The given rx descriptor.
- buffer1 – The first buffer address in the descriptor.
- buffer2 – The second buffer address in the descriptor.
- intEnable – Interrupt enable flag.
- doubleBuffEnable – The double buffer enable flag.

```
status_t ENET_QOS_ConfigureRxParser(ENET_QOS_Type *base, enet_qos_rxp_config_t  
                                   *rxpConfig, uint16_t entryCount)
```

Configure flexible rx parser.

This function is used to configure the flexible rx parser table.

Parameters

- base – ENET peripheral base address..
- rxpConfig – The rx parser configuration pointer.
- entryCount – The rx parser entry count.

Return values

- kStatus_Success – Configure rx parser success.
- kStatus_ENET_QOS_Timeout – Poll status flag timeout.

status_t ENET_QOS_ReadRxParser(ENET_QOS_Type *base, *enet_qos_rxp_config_t* *rxpConfig, uint16_t entryIndex)

Read flexible rx parser configuration at specified index.

This function is used to read flexible rx parser configuration at specified index.

Parameters

- base – ENET peripheral base address..
- rxpConfig – The rx parser configuration pointer.
- entryIndex – The rx parser entry index to read, start from 0.

Return values

- kStatus_Success – Configure rx parser success.
- kStatus_ENET_QOS_Timeout – Poll status flag timeout.

status_t ENET_QOS_EstProgramGcl(ENET_QOS_Type *base, *enet_qos_est_gcl_t* *gcl, uint32_t ptpClk_Hz)

Program Gate Control List.

This function is used to program the Enhanced Scheduled Transmission. (IEEE802.1Qbv)

Parameters

- base – ENET peripheral base address..
- gcl – Pointer to the Gate Control List structure.
- ptpClk_Hz – frequency of the PTP clock.

status_t ENET_QOS_EstReadGcl(ENET_QOS_Type *base, *enet_qos_est_gcl_t* *gcl, uint32_t listLen, bool hwList)

Read Gate Control List.

This function is used to read the Enhanced Scheduled Transmission list. (IEEE802.1Qbv)

Parameters

- base – ENET peripheral base address..
- gcl – Pointer to the Gate Control List structure.
- listLen – length of the provided opList array in gcl structure.
- hwList – Boolean if True read HW list, false read SW list.

static inline void ENET_QOS_FpeEnable(ENET_QOS_Type *base)

Enable Frame Preemption.

This function is used to enable frame preemption. (IEEE802.1Qbu)

Parameters

- base – ENET peripheral base address..

```
static inline void ENET_QOS_FpeDisable(ENET_QOS_Type *base)
```

Disable Frame Preemption.

This function is used to disable frame preemption. (IEEE802.1Qbu)

Parameters

- base – ENET peripheral base address..

```
static inline void ENET_QOS_FpeConfigPreemptable(ENET_QOS_Type *base, uint8_t  
queueMask)
```

Configure preemptable transmit queues.

This function is used to configure the preemptable queues. (IEEE802.1Qbu)

Parameters

- base – ENET peripheral base address..
- queueMask – bitmask representing queues to set in preemptable mode. The N-th bit represents the queue N.

```
void ENET_QOS_AVBConfigure(ENET_QOS_Type *base, const enet_qos_cbs_config_t *config,  
uint8_t queueIndex)
```

Sets the ENET AVB feature.

ENET_QOS AVB feature configuration, set transmit bandwidth. This API is called when the AVB feature is required.

Parameters

- base – ENET_QOS peripheral base address.
- config – The ENET_QOS AVB feature configuration structure.
- queueIndex – ENET_QOS queue index.

```
void ENET_QOS_GetStatistics(ENET_QOS_Type *base, enet_qos_transfer_stats_t *statistics)
```

Gets statistical data in transfer.

Parameters

- base – ENET_QOS peripheral base address.
- statistics – The statistics structure pointer.

```
void ENET_QOS_CreateHandler(ENET_QOS_Type *base, enet_qos_handle_t *handle,  
enet_qos_config_t *config, enet_qos_buffer_config_t  
*bufferConfig, enet_qos_callback_t callback, void *userData)
```

Create ENET Handler.

This is a transactional API and it's provided to store all data which are needed during the whole transactional process. This API should not be used when you use functional APIs to do data tx/rx. This is function will store many data/flag for transactional use, so all configure API such as ENET_QOS_Init(), ENET_QOS_DescriptorInit(), ENET_QOS_EnableInterrupts() etc.

Note: as our transactional transmit API use the zero-copy transmit buffer. so there are two thing we emphasize here:

- a. tx buffer free/requeue for application should be done in the tx interrupt handler. Please set callback: kENET_QOS_TxIntEvent with tx buffer free/requeue process APIs.
 - b. the tx interrupt is forced to open.
-

Parameters

- base – ENET peripheral base address.
- handle – ENET handler.
- config – ENET configuration.
- bufferConfig – ENET buffer configuration.
- callback – The callback function.
- userData – The application data.

status_t ENET_QOS_GetRxFramSize(ENET_QOS_Type *base, *enet_qos_handle_t* *handle, uint32_t *length, uint8_t channel)

Gets the size of the read frame. 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_QOS_GetRxFramSize, ENET_QOS_ReadFrame() should be called to update the receive buffers. If the result is not “kStatus_ENET_QOS_RxFramEmpty”.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler structure. This is the same handler pointer used in the ENET_QOS_Init.
- length – The length of the valid frame received.
- channel – The DMAC channel for the rx.

Return values

- kStatus_ENET_QOS_RxFramEmpty – No frame received. Should not call ENET_QOS_ReadFrame to read frame.
- kStatus_ENET_QOS_RxFramError – Data error happens. ENET_QOS_ReadFrame should be called with NULL data and NULL length to update the receive buffers.
- kStatus_Success – Receive a frame Successfully then the ENET_QOS_ReadFrame should be called with the right data buffer and the captured data length input.

status_t ENET_QOS_ReadFrame(ENET_QOS_Type *base, *enet_qos_handle_t* *handle, uint8_t *data, uint32_t length, uint8_t channel, *enet_qos_ptp_time_t* *ts)

Reads a frame from the ENET device. This function reads a frame from the ENET DMA descriptors. The ENET_QOS_GetRxFramSize should be used to get the size of the prepared data buffer. For example use rx dma channel 0:

```
uint32_t length;
enet_qos_handle_t g_handle;
status = ENET_QOS_GetRxFramSize(&g_handle, &length, 0);
if (length != 0)
{
    uint8_t *data = memory allocate interface;
    if (!data)
    {
        ENET_QOS_ReadFrame(ENET, &g_handle, NULL, 0, 0);
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    else
    {
        status = ENET_QOS_ReadFrame(ENET, &g_handle, data, length, 0);
    }
}
else if (status == kStatus_ENET_QOS_RxFrameError)
{
    ENET_QOS_ReadFrame(ENET, &g_handle, NULL, 0, 0);
}

```

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler structure. This is the same handler pointer used in the ENET_QOS_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.
- channel – The rx DMA channel. shall not be larger than 2.
- ts – Pointer to the structure enet_qos_ptp_time_t to save frame timestamp.

Returns

The execute status, successful or failure.

status_t ENET_QOS_SendFrame(ENET_QOS_Type *base, *enet_qos_handle_t* *handle, uint8_t *data, uint32_t length, uint8_t channel, bool isNeedTs, void *context, *enet_qos_tx_offload_t* txOffloadOps)

Transmits an ENET frame.

Note: The CRC is automatically appended to the data. Input the data to send without the CRC.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_QOS_Init.
- data – The data buffer provided by user to be send.
- length – The length of the data to be send.
- channel – Channel to send the frame, same with queue index.
- isNeedTs – True to enable timestamp save for the frame
- context – pointer to user context to be kept in the tx dirty frame information.
- txOffloadOps – The Tx frame checksum offload option.

Return values

- kStatus_Success – Send frame succeed.

- `kStatus_ENET_QOS_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_QOS_TxFrameBusy`.

`void ENET_QOS_ReclaimTxDescriptor(ENET_QOS_Type *base, enet_qos_handle_t *handle, uint8_t channel)`

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_QOS_Init`.
- `channel` – The tx DMA channel.

`void ENET_QOS_CommonIRQHandler(ENET_QOS_Type *base, enet_qos_handle_t *handle)`

The ENET IRQ handler.

Parameters

- `base` – ENET peripheral base address.
- `handle` – The ENET handler pointer.

`void ENET_QOS_SetISRHandler(ENET_QOS_Type *base, enet_qos_isr_t ISRHandler)`

Set the second level IRQ handler, allow user to overwrite the default second level weak IRQ handler.

Parameters

- `base` – ENET peripheral base address.
- `ISRHandler` – The handler to install.

`status_t ENET_QOS_Ptp1588CorrectTimerInCoarse(ENET_QOS_Type *base, enet_qos_systime_op operation, uint32_t second, uint32_t nanosecond)`

Correct the ENET PTP 1588 timer in coarse method.

Parameters

- `base` – ENET peripheral base address.
- `operation` – The system time operation, refer to “`enet_qos_systime_op`”
- `second` – The correction second.
- `nanosecond` – The correction nanosecond.

`status_t ENET_QOS_Ptp1588CorrectTimerInFine(ENET_QOS_Type *base, uint32_t addend)`

Correct the ENET PTP 1588 timer in fine method.

Note: Should take refer to the chapter “System time correction” and see the description for the “fine correction method”.

Parameters

- `base` – ENET peripheral base address.
- `addend` – The addend value to be set in the fine method


```
static inline uint32_t ENET_QOS_Ptp1588GetAddend(ENET_QOS_Type *base)
```

Get the ENET Time stamp current addend value.

Parameters

- base – ENET peripheral base address.

Returns

The addend value.

```
void ENET_QOS_Ptp1588GetTimerNoIRQDisable(ENET_QOS_Type *base, uint64_t *second,
                                           uint32_t *nanosecond)
```

Gets the current ENET time from the PTP 1588 timer without IRQ disable.

Parameters

- base – ENET peripheral base address.
- second – The PTP 1588 system timer second.
- nanosecond – The PTP 1588 system timer nanosecond. For the unit of the nanosecond is 1ns. so the nanosecond is the real nanosecond.

```
static inline status_t ENET_Ptp1588PpsControl(ENET_QOS_Type *base,
                                              enet_qos_ptp_pps_instance_t instance,
                                              enet_qos_ptp_pps_trgt_mode_t trgtMode,
                                              enet_qos_ptp_pps_cmd_t cmd)
```

Sets the ENET PTP 1588 PPS control. All channels operate in flexible PPS output mode.

Parameters

- base – ENET peripheral base address.
- instance – The ENET QOS PTP PPS instance.
- trgtMode – The target time register mode.
- cmd – The target flexible PPS output control command.

```
status_t ENET_QOS_Ptp1588PpsSetTrgtTime(ENET_QOS_Type *base,
                                          enet_qos_ptp_pps_instance_t instance, uint32_t
                                          seconds, uint32_t nanoseconds)
```

Sets the ENET QOS PTP 1588 PPS target time registers.

Parameters

- base – ENET QOS peripheral base address.
- instance – The ENET QOS PTP PPS instance.
- seconds – The target seconds.
- nanoseconds – The target nanoseconds.

```
static inline void ENET_QOS_Ptp1588PpsSetWidth(ENET_QOS_Type *base,
                                                enet_qos_ptp_pps_instance_t instance,
                                                uint32_t width)
```

Sets the ENET QOS PTP 1588 PPS output signal interval.

Parameters

- base – ENET QOS peripheral base address.
- instance – The ENET QOS PTP PPS instance.
- width – Signal Width. It is stored in terms of number of units of sub-second increment value. The width value must be lesser than interval value.

```
static inline void ENET_QOS_Ptp1588PpsSetInterval(ENET_QOS_Type *base,  
                                                enet_qos_ptp_pps_instance_t instance,  
                                                uint32_t interval)
```

Sets the ENET QOS PTP 1588 PPS output signal width.

Parameters

- base – ENET QOS peripheral base address.
- instance – The ENET QOS PTP PPS instance.
- interval – Signal Interval. It is stored in terms of number of units of sub-second increment value.

```
void ENET_QOS_Ptp1588GetTimer(ENET_QOS_Type *base, uint64_t *second, uint32_t  
                             *nanosecond)
```

Gets the current ENET time from the PTP 1588 timer.

Parameters

- base – ENET peripheral base address.
- second – The PTP 1588 system timer second.
- nanosecond – The PTP 1588 system timer nanosecond. For the unit of the nanosecond is 1ns.so the nanosecond is the real nanosecond.

```
void ENET_QOS_GetTxFrame(enet_qos_handle_t *handle, enet_qos_frame_info_t *txFrame,  
                        uint8_t channel)
```

Gets the time stamp of the transmit frame.

This function is used for PTP stack to get the timestamp captured by the ENET driver.

Parameters

- handle – The ENET handler pointer.This is the same state pointer used in ENET_QOS_Init.
- txFrame – Input parameter, pointer to *enet_qos_frame_info_t* for saving read out frame information.
- channel – Channel for searching the tx frame.

```
status_t ENET_QOS_GetRxFrame(ENET_QOS_Type *base, enet_qos_handle_t *handle,  
                             enet_qos_rx_frame_struct_t *rxFrame, uint8_t channel)
```

Receives one frame in specified BD ring with zero copy.

This function will use the user-defined allocate and free callback. Every time application gets one frame through this function, driver will allocate new buffers for the BDs whose buffers have been taken by application.

Note: This function will drop current frame and update related BDs as available for DMA if new buffers allocating fails. Application must provide a memory pool including at least BD number + 1 buffers(+2 if enable double buffer) to make this function work normally. If user calls this function in Rx interrupt handler, be careful that this function makes Rx BD ready with allocating new buffer(normal) or updating current BD(out of memory). If there's always new Rx frame input, Rx interrupt will be triggered forever. Application need to disable Rx interrupt according to specific design in this 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.
- channel – Channel for searching the rx frame.

Return values

- kStatus_Success – Succeed to get one frame and allocate new memory for Rx buffer.
- kStatus_ENET_QOS_RxFrameEmpty – There's no Rx frame in the BD.
- kStatus_ENET_QOS_RxFrameError – There's issue in this receiving.
- kStatus_ENET_QOS_RxFrameDrop – There's no new buffer memory for BD, drop this frame.

FSL_ENET_QOS_DRIVER_VERSION

Defines the driver version.

ENET_QOS_RXDESCRIP_RD_BUFF1VALID_MASK

Defines for read format.

Buffer1 address valid.

ENET_QOS_RXDESCRIP_RD_BUFF2VALID_MASK

Buffer2 address valid.

ENET_QOS_RXDESCRIP_RD_IOC_MASK

Interrupt enable on complete.

ENET_QOS_RXDESCRIP_RD_OWN_MASK

Own bit.

ENET_QOS_RXDESCRIP_WR_ERR_MASK

Defines for write back format.

ENET_QOS_RXDESCRIP_WR_PYLOAD_MASK

ENET_QOS_RXDESCRIP_WR_PTPMSGTYPE_MASK

ENET_QOS_RXDESCRIP_WR_PTPTYPE_MASK

ENET_QOS_RXDESCRIP_WR_PTPVERSION_MASK

ENET_QOS_RXDESCRIP_WR_PTPTSA_MASK

ENET_QOS_RXDESCRIP_WR_PACKETLEN_MASK

ENET_QOS_RXDESCRIP_WR_ERRSUM_MASK

ENET_QOS_RXDESCRIP_WR_TYPE_MASK

ENET_QOS_RXDESCRIP_WR_DE_MASK

ENET_QOS_RXDESCRIP_WR_RE_MASK

ENET_QOS_RXDESCRIP_WR_OE_MASK

ENET_QOS_RXDESCRIP_WR_RWT_MASK

ENET_QOS_RXDESCRIP_WR_GP_MASK

ENET_QOS_RXDESCRIP_WR_CRC_MASK

ENET_QOS_RXDESCRIP_WR_RS0V_MASK

ENET_QOS_RXDESCRIP_WR_RS1V_MASK

ENET_QOS_RXDESCRIP_WR_RS2V_MASK
ENET_QOS_RXDESCRIP_WR_LD_MASK
ENET_QOS_RXDESCRIP_WR_FD_MASK
ENET_QOS_RXDESCRIP_WR_CTXT_MASK
ENET_QOS_RXDESCRIP_WR_OWN_MASK
ENET_QOS_RXDESCRIP_WR_SA_FAILURE_MASK
ENET_QOS_RXDESCRIP_WR_DA_FAILURE_MASK
ENET_QOS_TXDESCRIP_RD_BL1_MASK

Defines for read format.

ENET_QOS_TXDESCRIP_RD_BL2_MASK
ENET_QOS_TXDESCRIP_RD_BL1(n)
ENET_QOS_TXDESCRIP_RD_BL2(n)
ENET_QOS_TXDESCRIP_RD_TTSE_MASK
ENET_QOS_TXDESCRIP_RD_IOC_MASK
ENET_QOS_TXDESCRIP_RD_FL_MASK
ENET_QOS_TXDESCRIP_RD_FL(n)
ENET_QOS_TXDESCRIP_RD_CIC(n)
ENET_QOS_TXDESCRIP_RD_TSE_MASK
ENET_QOS_TXDESCRIP_RD_SLOT(n)
ENET_QOS_TXDESCRIP_RD_SAIC(n)
ENET_QOS_TXDESCRIP_RD_CPC(n)
ENET_QOS_TXDESCRIP_RD_LDFD(n)
ENET_QOS_TXDESCRIP_RD_LD_MASK
ENET_QOS_TXDESCRIP_RD_FD_MASK
ENET_QOS_TXDESCRIP_RD_CTXT_MASK
ENET_QOS_TXDESCRIP_RD_OWN_MASK
ENET_QOS_TXDESCRIP_WB_TTSS_MASK

Defines for write back format.

ENET_QOS_ABNORM_INT_MASK
ENET_QOS_NORM_INT_MASK
ENET_QOS_RING_NUM_MAX
The Maximum number of tx/rx descriptor rings.
ENET_QOS_FRAME_MAX_FRAMELEN
Default maximum Ethernet frame size.

ENET_QOS_FCS_LEN

Ethernet FCS length.

ENET_QOS_ADDR_ALIGNMENT

Recommended Ethernet buffer alignment.

ENET_QOS_BUFF_ALIGNMENT

Receive buffer alignment shall be 4bytes-aligned.

ENET_QOS_MTL_RXFIFOSIZE

The rx fifo size.

ENET_QOS_MTL_TXFIFOSIZE

The tx fifo size.

ENET_QOS_MACINT_ENUM_OFFSET

The offset for mac interrupt in enum type.

ENET_QOS_RXP_ENTRY_COUNT

RXP table entry count, implied by FRPES in MAC_HW_FEATURE3

ENET_QOS_RXP_BUFFER_SIZE

RXP Buffer size, implied by FRPBS in MAC_HW_FEATURE3

ENET_QOS_EST_WID

Width of the time interval in Gate Control List

ENET_QOS_EST_DEP

Maximum depth of Gate Control List

Defines the status return codes for transaction.

Values:

enumerator kStatus_ENET_QOS_InitMemoryFail

Init fails since buffer memory is not enough.

enumerator kStatus_ENET_QOS_RxFrameError

A frame received but data error happen.

enumerator kStatus_ENET_QOS_RxFrameFail

Failed to receive a frame.

enumerator kStatus_ENET_QOS_RxFrameEmpty

No frame arrive.

enumerator kStatus_ENET_QOS_RxFrameDrop

Rx frame is dropped since no buffer memory.

enumerator kStatus_ENET_QOS_TxFrameBusy

Transmit descriptors are under process.

enumerator kStatus_ENET_QOS_TxFrameFail

Transmit frame fail.

enumerator kStatus_ENET_QOS_TxFrameOverLen

Transmit oversize.

enumerator kStatus_ENET_QOS_Est_SwListBusy

SW Gcl List not yet processed by HW.

enumerator kStatus_ENET_QOS_Est_SwListWriteAbort
SW Gcl List write aborted .

enumerator kStatus_ENET_QOS_Est_InvalidParameter
Invalid parameter in Gcl List .

enumerator kStatus_ENET_QOS_Est_BtrError
Base Time Error when loading list.

enumerator kStatus_ENET_QOS_TrgtBusy
Target time register busy.

enumerator kStatus_ENET_QOS_Timeout
Target time register busy.

enumerator kStatus_ENET_QOS_PpsBusy
Pps command busy.

enum _enet_qos_mii_mode

Defines the MII/RGMII mode for data interface between the MAC and the PHY.

Values:

enumerator kENET_QOS_MiiMode
MII mode for data interface.

enumerator kENET_QOS_RgmiiMode
RGMII mode for data interface.

enumerator kENET_QOS_RmiiMode
RMII mode for data interface.

enum _enet_qos_mii_speed

Defines the 10/100/1000 Mbps speed for the MII data interface.

Values:

enumerator kENET_QOS_MiiSpeed10M
Speed 10 Mbps.

enumerator kENET_QOS_MiiSpeed100M
Speed 100 Mbps.

enumerator kENET_QOS_MiiSpeed1000M
Speed 1000 Mbps.

enumerator kENET_QOS_MiiSpeed2500M
Speed 2500 Mbps.

enum _enet_qos_mii_duplex

Defines the half or full duplex for the MII data interface.

Values:

enumerator kENET_QOS_MiiHalfDuplex
Half duplex mode.

enumerator kENET_QOS_MiiFullDuplex
Full duplex mode.

enum _enet_qos_mii_normal_opcode

Define the MII opcode for normal MDIO_CLAUSES_22 Frame.

Values:

enumerator kENET_QOS_MiiWriteFrame

Write frame operation for a valid MII management frame.

enumerator kENET_QOS_MiiReadFrame

Read frame operation for a valid MII management frame.

enum _enet_qos_dma_burstlen

Define the DMA maximum transmit burst length.

Values:

enumerator kENET_QOS_BurstLen1

DMA burst length 1.

enumerator kENET_QOS_BurstLen2

DMA burst length 2.

enumerator kENET_QOS_BurstLen4

DMA burst length 4.

enumerator kENET_QOS_BurstLen8

DMA burst length 8.

enumerator kENET_QOS_BurstLen16

DMA burst length 16.

enumerator kENET_QOS_BurstLen32

DMA burst length 32.

enumerator kENET_QOS_BurstLen64

DMA burst length 64. eight times enabled.

enumerator kENET_QOS_BurstLen128

DMA burst length 128. eight times enabled.

enumerator kENET_QOS_BurstLen256

DMA burst length 256. eight times enabled.

enum _enet_qos_desc_flag

Define the flag for the descriptor.

Values:

enumerator kENET_QOS_MiddleFlag

It's a middle descriptor of the frame.

enumerator kENET_QOS_LastFlagOnly

It's the last descriptor of the frame.

enumerator kENET_QOS_FirstFlagOnly

It's the first descriptor of the frame.

enumerator kENET_QOS_FirstLastFlag

It's the first and last descriptor of the frame.

enum _enet_qos_systime_op

Define the system time adjust operation control.

Values:

enumerator kENET_QOS_SystimeAdd

System time add to.

enumerator kENET_QOS_SystimeSubtract
System time subtract.

enum _enet_qos_ts_rollover_type
Define the system time rollover control.

Values:

enumerator kENET_QOS_BinaryRollover
System time binary rollover.

enumerator kENET_QOS_DigitalRollover
System time digital rollover.

enum _enet_qos_special_config
Defines some special configuration for ENET.

These control flags are provided for special user requirements. Normally, there is no need to set these control flags for ENET initialization. But if you have some special requirements, set the flags to specialControl in the enet_qos_config_t.

Note: “kENET_QOS_StoreAndForward” is recommended to be set.

Values:

enumerator kENET_QOS_DescDoubleBuffer
The double buffer is used in the tx/rx descriptor.

enumerator kENET_QOS_StoreAndForward
The rx/tx store and forward enable.

enumerator kENET_QOS_PromiscuousEnable
The promiscuous enabled.

enumerator kENET_QOS_FlowControlEnable
The flow control enabled.

enumerator kENET_QOS_BroadCastRxDisable
The broadcast disabled.

enumerator kENET_QOS_MulticastAllEnable
All multicast are passed.

enumerator kENET_QOS_8023AS2KPacket
8023as support for 2K packets.

enumerator kENET_QOS_HashMulticastEnable
The multicast packets are filtered through hash table.

enumerator kENET_QOS_RxChecksumOffloadEnable
The Rx checksum offload enabled.

enum _enet_qos_dma_interrupt_enable
List of DMA interrupts supported by the ENET interrupt. This enumeration uses one-bit encoding to allow a logical OR of multiple members.

Values:

enumerator kENET_QOS_DmaTx
Tx interrupt.

enumerator kENET_QOS_DmaTxStop

Tx stop interrupt.

enumerator kENET_QOS_DmaTxBuffUnavail

Tx buffer unavailable.

enumerator kENET_QOS_DmaRx

Rx interrupt.

enumerator kENET_QOS_DmaRxBuffUnavail

Rx buffer unavailable.

enumerator kENET_QOS_DmaRxStop

Rx stop.

enumerator kENET_QOS_DmaRxWatchdogTimeout

Rx watchdog timeout.

enumerator kENET_QOS_DmaEarlyTx

Early transmit.

enumerator kENET_QOS_DmaEarlyRx

Early receive.

enumerator kENET_QOS_DmaBusErr

Fatal bus error.

enum _enet_qos_mac_interrupt_enable

List of mac interrupts supported by the ENET interrupt. This enumeration uses one-bot encoding to allow a logical OR of multiple members.

Values:

enumerator kENET_QOS_MacTimestamp

enum _enet_qos_event

Defines the common interrupt event for callback use.

Values:

enumerator kENET_QOS_RxIntEvent

Receive interrupt event.

enumerator kENET_QOS_TxIntEvent

Transmit interrupt event.

enumerator kENET_QOS_WakeUpIntEvent

Wake up interrupt event.

enumerator kENET_QOS_TimeStampIntEvent

Time stamp interrupt event.

enum _enet_qos_queue_mode

Define the MTL mode for multiple queues/rings.

Values:

enumerator kENET_QOS_AVB_Mode

Enable queue in AVB mode.

enumerator kENET_QOS_DCB_Mode

Enable queue in DCB mode.

enum _enet_qos_mtl_multiqueue_txsche

Define the MTL tx scheduling algorithm for multiple queues/rings.

Values:

enumerator kENET_QOS_txWeightRR

Tx weight round-robin.

enumerator kENET_QOS_txWeightFQ

Tx weight fair queuing.

enumerator kENET_QOS_txDeficitWeightRR

Tx deficit weighted round-robin.

enumerator kENET_QOS_txStrPrio

Tx strict priority.

enum _enet_qos_mtl_multiqueue_rxsche

Define the MTL rx scheduling algorithm for multiple queues/rings.

Values:

enumerator kENET_QOS_rxStrPrio

Rx strict priority, Queue 0 has the lowest priority.

enumerator kENET_QOS_rxWeightStrPrio

Weighted Strict Priority.

enum _enet_qos_mtl_rxqueuemap

Define the MTL rx queue and DMA channel mapping.

Values:

enumerator kENET_QOS_StaticDirctMap

The received frame in rx Qn(n = 0,1) directly map to dma channel n.

enumerator kENET_QOS_DynamicMap

The received frame in rx Qn(n = 0,1) map to the dma channel m(m = 0,1) related with the same Mac.

enum _enet_qos_rx_queue_route

Defines the package type for receive queue routing.

Values:

enumerator kENET_QOS_PacketNoQ

enumerator kENET_QOS_PacketAVCPQ

enumerator kENET_QOS_PacketPTPQ

enumerator kENET_QOS_PacketUPQ

enumerator kENET_QOS_PacketMCBCQ

enum _enet_qos_ptp_event_type

Defines the ENET PTP message related constant.

Values:

enumerator kENET_QOS_PtpEventMsgType

PTP event message type.

enumerator kENET_QOS_PtpSrcPortIdLen

PTP message sequence id length.

enumerator kENET_QOS_PtpEventPort

PTP event port number.

enumerator kENET_QOS_PtpGnrlPort

PTP general port number.

enum _enet_qos_ptp_pps_instance

Defines the PPS instance numbers.

Values:

enumerator kENET_QOS_PtpPpsInstance0

PPS instance 0.

enumerator kENET_QOS_PtpPpsInstance1

PPS instance 1.

enumerator kENET_QOS_PtpPpsInstance2

PPS instance 2.

enumerator kENET_QOS_PtpPpsInstance3

PPS instance 3.

enum _enet_qos_ptp_pps_trgt_mode

Defines the Target Time register mode.

Values:

enumerator kENET_QOS_PtpPpsTrgtModeOnlyInt

Only interrupts.

enumerator kENET_QOS_PtpPpsTrgtModeIntSt

Both interrupt and output signal.

enumerator kENET_QOS_PtpPpsTrgtModeOnlySt

Only output signal.

enum _enet_qos_ptp_pps_cmd

Defines commands for ppscmd register.

Values:

enumerator kENET_QOS_PtpPpsCmdNC

No Command.

enumerator kENET_QOS_PtpPpsCmdSSP

Start Single Pulse.

enumerator kENET_QOS_PtpPpsCmdSPT

Start Pulse Train.

enumerator kENET_QOS_PtpPpsCmdCS

Cancel Start.

enumerator kENET_QOS_PtpPpsCmdSPTAT

Stop Pulse Train At Time.

enumerator kENET_QOS_PtpPpsCmdSPTI

Stop Pulse Train Immediately.

enumerator kENET_QOS_PtpPpsCmdCSPT

Cancel Stop Pulse Train.

enum _enet_qos_ets_list_length

Defines the enumeration of ETS list length.

Values:

enumerator kENET_QOS_Ets_List_64

List length of 64

enumerator kENET_QOS_Ets_List_128

List length of 128

enumerator kENET_QOS_Ets_List_256

List length of 256

enumerator kENET_QOS_Ets_List_512

List length of 512

enumerator kENET_QOS_Ets_List_1024

List length of 1024

enum _enet_qos_ets_gccr_addr

Defines the enumeration of ETS gate control address.

Values:

enumerator kENET_QOS_Ets_btr_low

BTR Low

enumerator kENET_QOS_Ets_btr_high

BTR High

enumerator kENET_QOS_Ets_ctr_low

CTR Low

enumerator kENET_QOS_Ets_ctr_high

CTR High

enumerator kENET_QOS_Ets_ter

TER

enumerator kENET_QOS_Ets_llr

LLR

enum _enet_qos_rxp_dma_chn

Defines the enumeration of DMA channel used for rx parser entry.

Values:

enumerator kENET_QOS_Rxp_DMACHn0

DMA Channel 0 used for RXP entry match

enumerator kENET_QOS_Rxp_DMACHn1

DMA Channel 1 used for RXP entry match

enumerator kENET_QOS_Rxp_DMACHn2

DMA Channel 2 used for RXP entry match

enumerator kENET_QOS_Rxp_DMACHn3

DMA Channel 3 used for RXP entry match

enumerator kENET_QOS_Rxp_DMACHn4

DMA Channel 4 used for RXP entry match

enum *_enet_qos_tx_offload*

Define the Tx checksum offload options.

Values:

enumerator *kENET_QOS_TxOffloadDisable*

Disable Tx checksum offload.

enumerator *kENET_QOS_TxOffloadIPHeader*

Enable IP header checksum calculation and insertion.

enumerator *kENET_QOS_TxOffloadIPHeaderPlusPayload*

Enable IP header and payload checksum calculation and insertion.

enumerator *kENET_QOS_TxOffloadAll*

Enable IP header, payload and pseudo header checksum calculation and insertion.

typedef enum *_enet_qos_mii_mode* *enet_qos_mii_mode_t*

Defines the MII/RGMII mode for data interface between the MAC and the PHY.

typedef enum *_enet_qos_mii_speed* *enet_qos_mii_speed_t*

Defines the 10/100/1000 Mbps speed for the MII data interface.

typedef enum *_enet_qos_mii_duplex* *enet_qos_mii_duplex_t*

Defines the half or full duplex for the MII data interface.

typedef enum *_enet_qos_mii_normal_opcode* *enet_qos_mii_normal_opcode*

Define the MII opcode for normal MDIO_CLAUSES_22 Frame.

typedef enum *_enet_qos_dma_burstlen* *enet_qos_dma_burstlen*

Define the DMA maximum transmit burst length.

typedef enum *_enet_qos_desc_flag* *enet_qos_desc_flag*

Define the flag for the descriptor.

typedef enum *_enet_qos_systime_op* *enet_qos_systime_op*

Define the system time adjust operation control.

typedef enum *_enet_qos_ts_rollover_type* *enet_qos_ts_rollover_type*

Define the system time rollover control.

typedef enum *_enet_qos_special_config* *enet_qos_special_config_t*

Defines some special configuration for ENET.

These control flags are provided for special user requirements. Normally, there is no need to set these control flags for ENET initialization. But if you have some special requirements, set the flags to specialControl in the *enet_qos_config_t*.

Note: “*kENET_QOS_StoreAndForward*” is recommended to be set.

typedef enum *_enet_qos_dma_interrupt_enable* *enet_qos_dma_interrupt_enable_t*

List of DMA interrupts supported by the ENET interrupt. This enumeration uses one-bit encoding to allow a logical OR of multiple members.

typedef enum *_enet_qos_mac_interrupt_enable* *enet_qos_mac_interrupt_enable_t*

List of mac interrupts supported by the ENET interrupt. This enumeration uses one-bit encoding to allow a logical OR of multiple members.

typedef enum *_enet_qos_event* *enet_qos_event_t*

Defines the common interrupt event for callback use.

typedef enum *_enet_qos_queue_mode* enet_qos_queue_mode_t

Define the MTL mode for multiple queues/rings.

typedef enum *_enet_qos_mtl_multiqueue_txsche* enet_qos_mtl_multiqueue_txsche

Define the MTL tx scheduling algorithm for multiple queues/rings.

typedef enum *_enet_qos_mtl_multiqueue_rxsche* enet_qos_mtl_multiqueue_rxsche

Define the MTL rx scheduling algorithm for multiple queues/rings.

typedef enum *_enet_qos_mtl_rxqueuemap* enet_qos_mtl_rxqueuemap_t

Define the MTL rx queue and DMA channel mapping.

typedef enum *_enet_qos_rx_queue_route* enet_qos_rx_queue_route_t

Defines the package type for receive queue routing.

typedef enum *_enet_qos_ptp_event_type* enet_qos_ptp_event_type_t

Defines the ENET PTP message related constant.

typedef enum *_enet_qos_ptp_pps_instance* enet_qos_ptp_pps_instance_t

Defines the PPS instance numbers.

typedef enum *_enet_qos_ptp_pps_trgt_mode* enet_qos_ptp_pps_trgt_mode_t

Defines the Target Time register mode.

typedef enum *_enet_qos_ptp_pps_cmd* enet_qos_ptp_pps_cmd_t

Defines commands for ppscmd register.

typedef enum *_enet_qos_ets_list_length* enet_qos_ets_list_length_t

Defines the enumeration of ETS list length.

typedef enum *_enet_qos_ets_gccr_addr* enet_qos_ets_gccr_addr_t

Defines the enumeration of ETS gate control address.

typedef enum *_enet_qos_rxp_dma_chn* enet_qos_rxp_dma_chn_t

Defines the enumeration of DMA channel used for rx parser entry.

typedef enum *_enet_qos_tx_offload* enet_qos_tx_offload_t

Define the Tx checksum offload options.

typedef struct *_enet_qos_rx_bd_struct* enet_qos_rx_bd_struct_t

Defines the receive descriptor structure has the read-format and write-back format structure. They both has the same size with different region definition. so we define the read-format region as the receive descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the receive data process.

typedef struct *_enet_qos_tx_bd_struct* enet_qos_tx_bd_struct_t

Defines the transmit descriptor structure has the read-format and write-back format structure. They both has the same size with different region definition. so we define the read-format region as the transmit descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the transmit data process.

typedef struct *_enet_qos_tx_bd_config_struct* enet_qos_tx_bd_config_struct_t

Defines the Tx BD configuration structure.

typedef struct *_enet_qos_ptp_time* enet_qos_ptp_time_t

Defines the ENET PTP time stamp structure.

typedef struct *enet_qos_frame_info* enet_qos_frame_info_t

Defines the frame info structure.

```
typedef struct _enet_qos_tx_dirty_ring enet_qos_tx_dirty_ring_t
```

Defines the ENET transmit dirty addresses ring/queue structure.

```
typedef struct _enet_qos_ptp_config enet_qos_ptp_config_t
```

Defines the ENET PTP configuration structure.

```
typedef struct _enet_qos_est_gate_op enet_qos_est_gate_op_t
```

Defines the EST gate operation structure.

```
typedef struct _enet_qos_est_gcl enet_qos_est_gcl_t
```

Defines the EST gate control list structure.

```
typedef struct _enet_qos_rxp_config enet_qos_rxp_config_t
```

Defines the ENET_QOS Rx parser configuration structure.

```
typedef struct _enet_qos_buffer_config enet_qos_buffer_config_t
```

Defines the buffer descriptor configure structure.

Note:

- a. The receive and transmit descriptor start address pointer and tail pointer must be word-aligned.
 - b. The recommended minimum tx/rx ring length is 4.
 - c. The tx/rx descriptor tail address shall be the address pointer to the address just after the end of the last last descriptor. because only the descriptors between the start address and the tail address will be used by DMA.
 - d. The descriptor address is the start address of all used contiguous memory. for example, the rxDescStartAddrAlign is the start address of rxRingLen contiguous descriptor memories for rx descriptor ring 0.
 - e. The “*rxBufferstartAddr” is the first element of rxRingLen (2*rxRingLen for double buffers) rx buffers. It means the *rxBufferStartAddr is the rx buffer for the first descriptor the *rxBufferStartAddr + 1 is the rx buffer for the second descriptor or the rx buffer for the second buffer in the first descriptor. so please make sure the rxBufferStartAddr is the address of a rxRingLen or 2*rxRingLen array.
-

```
typedef struct _enet_qos_cbs_config enet_qos_cbs_config_t
```

Defines the CBS configuration for queue.

```
typedef struct enet_qos_tx_queue_config enet_qos_queue_tx_config_t
```

Defines the queue configuration structure.

```
typedef struct enet_qos_rx_queue_config enet_qos_queue_rx_config_t
```

Defines the queue configuration structure.

```
typedef struct enet_qos_multiqueue_config enet_qos_multiqueue_config_t
```

Defines the configuration when multi-queue is used.

```
typedef void (*enet_qos_rx_alloc_callback_t)(ENET_QOS_Type *base, void *userData, uint8_t channel)
```

Defines the Rx memory buffer alloc function pointer.

```
typedef void (*enet_qos_rx_free_callback_t)(ENET_QOS_Type *base, void *buffer, void *userData, uint8_t channel)
```

Defines the Rx memory buffer free function pointer.

```
typedef struct _enet_qos_config enet_qos_config_t
```

Defines the basic configuration structure for the ENET device.

Note: Default the signal queue is used so the “*multiqueueCfg” is set default with NULL. Set the pointer with a valid configuration pointer if the multiple queues are required. If multiple queue is enabled, please make sure the buffer configuration for all are prepared also.

```
typedef struct _enet_qos_handle enet_qos_handle_t
```

```
typedef void (*enet_qos_callback_t)(ENET_QOS_Type *base, enet_qos_handle_t *handle,  
enet_qos_event_t event, uint8_t channel, void *userData)
```

ENET callback function.

```
typedef struct _enet_qos_tx_bd_ring enet_qos_tx_bd_ring_t
```

Defines the ENET transmit buffer descriptor ring/queue structure.

```
typedef struct _enet_qos_rx_bd_ring enet_qos_rx_bd_ring_t
```

Defines the ENET receive buffer descriptor ring/queue structure.

```
typedef struct _enet_qos_state enet_qos_state_t
```

Defines the ENET state structure.

Note: The structure contains saved state for the instance. It could be stored in `enet_qos_handle_t`, but that's used only with the transactional API.

```
typedef struct _enet_qos_buffer_struct enet_qos_buffer_struct_t
```

Defines the frame buffer structure.

```
typedef struct _enet_qos_rx_frame_error enet_qos_rx_frame_error_t
```

Defines the Rx frame error structure.

```
typedef struct _enet_qos_rx_frame_attribute_struct enet_qos_rx_frame_attribute_t
```

```
typedef struct _enet_qos_rx_frame_struct enet_qos_rx_frame_struct_t
```

Defines the Rx frame data structure.

```
typedef struct _enet_qos_transfer_stats enet_qos_transfer_stats_t
```

Defines the ENET QOS transfer statistics structure.

```
typedef void (*enet_qos_isr_t)(ENET_QOS_Type *base, enet_qos_handle_t *handle)
```

```
const clock_ip_name_t s_enetqosClock[]
```

Pointers to enet clocks for each instance.

```
void ENET_QOS_SetSYSControl(enet_qos_mii_mode_t miiMode)
```

Set ENET system configuration.

Note: User needs to provide the implementation because the implementation is SoC specific. This function set the phy selection and enable clock. It should be called before any other ethernet operation.

Parameters

- `miiMode` – The MII/RGMII/RMII mode for interface between the phy and Ethernet.


```
void ENET_QOS_EnableClock(bool enable)
```

Enable/Disable ENET qos clock.

Note: User needs to provide the implementation because the implementation is SoC specific. This function should be called before config RMII mode.

```
struct __enet_qos_rx_bd_struct
```

#include <fsl_enet_qos.h> Defines the receive descriptor structure has the read-format and write-back format structure. They both has the same size with different region definition. so we define the read-format region as the receive descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the receive data process.

Public Members

```
__IO uint32_t buff1Addr
```

Buffer 1 address

```
__IO uint32_t reserved
```

Reserved

```
__IO uint32_t buff2Addr
```

Buffer 2 or next descriptor address

```
__IO uint32_t control
```

Buffer 1/2 byte counts and control

```
struct __enet_qos_tx_bd_struct
```

#include <fsl_enet_qos.h> Defines the transmit descriptor structure has the read-format and write-back format structure. They both has the same size with different region definition. so we define the read-format region as the transmit descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the transmit data process.

Public Members

```
__IO uint32_t buff1Addr
```

Buffer 1 address

```
__IO uint32_t buff2Addr
```

Buffer 2 address

```
__IO uint32_t buffLen
```

Buffer 1/2 byte counts

```
__IO uint32_t controlStat
```

TDES control and status word

```
struct __enet_qos_tx_bd_config_struct
```

#include <fsl_enet_qos.h> Defines the Tx BD configuration structure.

Public Members

```
void *buffer1
```

The first buffer address in the descriptor.

uint32_t bytes1

The bytes in the first buffer.

void *buffer2

The second buffer address in the descriptor.

uint32_t bytes2

The bytes in the second buffer.

uint32_t framelen

The length of the frame to be transmitted.

bool intEnable

Interrupt enable flag.

bool tsEnable

The timestamp enable.

enet_qos_tx_offload_t txOffloadOps

The Tx checksum offload option.

enet_qos_desc_flag flag

The flag of this tx descriptor, see “enet_qos_desc_flag”.

struct _enet_qos_ptp_time

#include <fsl_enet_qos.h> Defines the ENET PTP time stamp structure.

Public Members

uint64_t second

Second.

uint32_t nanosecond

Nanosecond.

struct enet_qos_frame_info

#include <fsl_enet_qos.h> Defines the frame info structure.

Public Members

void *context

User specified data, could be buffer address for free

bool isTsAvail

Flag indicates timestamp available status

enet_qos_ptp_time_t timeStamp

Timestamp of frame

struct _enet_qos_tx_dirty_ring

#include <fsl_enet_qos.h> Defines the ENET transmit dirty addresses ring/queue structure.

Public Members

enet_qos_frame_info_t *txDirtyBase

Dirty buffer descriptor base address pointer.

uint16_t txGenIdx

tx generate index.

uint16_t txConsumIdx
tx consume index.

uint16_t txRingLen
tx ring length.

bool isFull
tx ring is full flag, add this parameter to avoid waste one element.

struct __enet_qos_ptp_config
#include <fsl_enet_qos.h> Defines the ENET PTP configuration structure.

Public Members

bool fineUpdateEnable
Use the fine update.

uint32_t defaultAddend
Default addend value when fine update is enable, could be $2^{32} / (\text{refClk\_Hz} / \text{ENET\_QOS\_MICRSECS\_ONESECOND} / \text{ENET\_QOS\_SYSTIME\_REQUIRED\_CLK\_MHZ})$.

bool ptp1588V2Enable
The desired system time frequency. Must be lower than reference clock. (Only used with fine correction method). ptp 1588 version 2 is used.

enet_qos_ts_rollover_type tsRollover
1588 time nanosecond rollover.

struct __enet_qos_est_gate_op
#include <fsl_enet_qos.h> Defines the EST gate operation structure.

struct __enet_qos_est_gcl
#include <fsl_enet_qos.h> Defines the EST gate control list structure.

Public Members

bool enable
Enable or disable EST

uint64_t cycleTime
Base Time 32 bits seconds 32 bits nanoseconds

uint32_t extTime
Cycle Time 32 bits seconds 32 bits nanoseconds

uint32_t numEntries
Time Extension 32 bits seconds 32 bits nanoseconds

enet_qos_est_gate_op_t *opList
Number of entries

struct __enet_qos_rxp_config
#include <fsl_enet_qos.h> Defines the ENET_QOS Rx parser configuration structure.

Public Members

uint32_t matchEnable
4-byte match data used for comparing with incoming packet

uint8_t acceptFrame

When matchEnable is set to 1, the matchData is used for comparing

uint8_t rejectFrame

When acceptFrame = 1 and data is matched, the frame will be sent to DMA channel

uint8_t inverseMatch

When rejectFrame = 1 and data is matched, the frame will be dropped

uint8_t nextControl

Inverse match

uint8_t reserved

Next instruction indexing control

uint8_t frameOffset

Reserved control fields

uint8_t okIndex

Frame offset in the packet data to be compared for match, in terms of 4 bytes.

uint8_t dmaChannel

Memory Index to be used next.

uint32_t reserved2

The DMA channel `enet_qos_rxp_dma_chn_t` used for receiving the frame when frame match and acceptFrame = 1

struct _enet_qos_buffer_config

#include <fsl_enet_qos.h> Defines the buffer descriptor configure structure.

Note:

- The receive and transmit descriptor start address pointer and tail pointer must be word-aligned.
 - The recommended minimum tx/rx ring length is 4.
 - The tx/rx descriptor tail address shall be the address pointer to the address just after the end of the last last descriptor. because only the descriptors between the start address and the tail address will be used by DMA.
 - The descriptor address is the start address of all used contiguous memory. for example, the `rxDescStartAddrAlign` is the start address of `rxRingLen` contiguous descriptor memories for rx descriptor ring 0.
 - The “`*rxBufferstartAddr`” is the first element of `rxRingLen` (`2*rxRingLen` for double buffers) rx buffers. It means the `*rxBufferStartAddr` is the rx buffer for the first descriptor the `*rxBufferStartAddr + 1` is the rx buffer for the second descriptor or the rx buffer for the second buffer in the first descriptor. so please make sure the `rxBufferStartAddr` is the address of a `rxRingLen` or `2*rxRingLen` array.
-

Public Members

uint8_t rxRingLen

The length of receive buffer descriptor ring.

uint8_t txRingLen

The length of transmit buffer descriptor ring.

enet_qos_tx_bd_struct_t *txDescStartAddrAlign

Aligned transmit descriptor start address.

enet_qos_tx_bd_struct_t *txDescTailAddrAlign

Aligned transmit descriptor tail address.

enet_qos_frame_info_t *txDirtyStartAddr

Start address of the dirty tx frame information.

enet_qos_rx_bd_struct_t *rxDescStartAddrAlign

Aligned receive descriptor start address.

enet_qos_rx_bd_struct_t *rxDescTailAddrAlign

Aligned receive descriptor tail address.

uint32_t *rxBufferStartAddr

Start address of the rx buffers.

uint32_t rxBuffSizeAlign

Aligned receive data buffer size.

bool rxBuffNeedMaintain

Whether receive data buffer need cache maintain.

struct __enet_qos_cbs_config

#include <fsl_enet_qos.h> Defines the CBS configuration for queue.

Public Members

uint16_t sendSlope

Send slope configuration.

uint16_t idleSlope

Idle slope configuration.

uint32_t highCredit

High credit.

uint32_t lowCredit

Low credit.

struct enet_qos_tx_queue_config

#include <fsl_enet_qos.h> Defines the queue configuration structure.

Public Members

enet_qos_queue_mode_t mode

tx queue mode configuration.

uint32_t weight

Refer to the MTL TxQ Quantum Weight register.

uint32_t priority

Refer to Transmit Queue Priority Mapping register.

enet_qos_cbs_config_t *cbsConfig

CBS configuration if queue use AVB mode.

struct enet_qos_rx_queue_config

#include <fsl_enet_qos.h> Defines the queue configuration structure.

Public Members

enet_qos_queue_mode_t mode
rx queue mode configuration.

uint8_t mapChannel
tx queue map dma channel.

uint32_t priority
Rx queue priority.

enet_qos_rx_queue_route_t packetRoute
Receive packet routing.

struct *enet_qos_multiqueue_config*
#include <fsl_enet_qos.h> Defines the configuration when multi-queue is used.

Public Members

enet_qos_dma_burstlen burstLen
Burst len for the multi-queue.

uint8_t txQueueUse
Used Tx queue count.

enet_qos_mtl_multiqueue_txsche mtltxSche
Transmit schedule for multi-queue.

enet_qos_queue_tx_config_t txQueueConfig[ENET_QOS_DMA_CH_COUNT]
Tx Queue configuration.

uint8_t rxQueueUse
Used Rx queue count.

enet_qos_mtl_multiqueue_rxsche mtlrxSche
Receive schedule for multi-queue.

enet_qos_queue_rx_config_t rxQueueConfig[ENET_QOS_DMA_CH_COUNT]
Rx Queue configuration.

struct *_enet_qos_config*
#include <fsl_enet_qos.h> Defines the basic configuration structure for the ENET device.

Note: Default the signal queue is used so the “*multiqueueCfg” is set default with NULL. Set the pointer with a valid configuration pointer if the multiple queues are required. If multiple queue is enabled, please make sure the buffer configuration for all are prepared also.

Public Members

uint16_t specialControl
The logic or of *enet_qos_special_config_t*

enet_qos_multiqueue_config_t *multiqueueCfg
Use multi-queue.

enet_qos_mii_mode_t miiMode
MII mode.

enet_qos_mii_speed_t miiSpeed

MII Speed.

enet_qos_mii_duplex_t miiDuplex

MII duplex.

uint16_t pauseDuration

Used in the tx flow control frame, only valid when kENET_QOS_FlowControlEnable is set.

enet_qos_ptp_config_t *ptpConfig

PTP 1588 feature configuration

uint32_t csrClock_Hz

CSR clock frequency in HZ.

enet_qos_rx_alloc_callback_t rxBuffAlloc

Callback to alloc memory, must be provided for zero-copy Rx.

enet_qos_rx_free_callback_t rxBuffFree

Callback to free memory, must be provided for zero-copy Rx.

struct *_enet_qos_tx_bd_ring*

#include <fsl_enet_qos.h> Defines the ENET transmit buffer descriptor ring/queue structure.

Public Members

enet_qos_tx_bd_struct_t *txBdBase

Buffer descriptor base address pointer.

uint16_t txGenIdx

tx generate index.

uint16_t txConsumIdx

tx consume index.

volatile *uint16_t* txDescUsed

tx descriptor used number.

uint16_t txRingLen

tx ring length.

struct *_enet_qos_rx_bd_ring*

#include <fsl_enet_qos.h> Defines the ENET receive buffer descriptor ring/queue structure.

Public Members

enet_qos_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.

uint32_t rxBuffSizeAlign

Receive buffer size.

struct *_enet_qos_handle*

#include <fsl_enet_qos.h> Defines the ENET handler structure.

Public Members

`uint8_t txQueueUse`
Used tx queue count.

`uint8_t rxQueueUse`
Used rx queue count.

`bool doubleBuffEnable`
The double buffer is used in the descriptor.

`bool rxintEnable`
Rx interrupt enabled.

`bool rxMaintainEnable[ENET_QOS_DMA_CH_COUNT]`
Rx buffer cache maintain enabled.

`enet_qos_rx_bd_ring_t rxBdRing[ENET_QOS_DMA_CH_COUNT]`
Receive buffer descriptor.

`enet_qos_tx_bd_ring_t txBdRing[ENET_QOS_DMA_CH_COUNT]`
Transmit buffer descriptor.

`enet_qos_tx_dirty_ring_t txDirtyRing[ENET_QOS_DMA_CH_COUNT]`
Transmit dirty buffers addresses.

`uint32_t *rxBufferStartAddr[ENET_QOS_DMA_CH_COUNT]`
Rx buffer start address for reInitialize.

`enet_qos_callback_t callback`
Callback function.

`void *userData`
Callback function parameter.

`uint8_t multicastCount[64]`
Multicast collisions counter

`enet_qos_rx_alloc_callback_t rxBuffAlloc`
Callback to alloc memory, must be provided for zero-copy Rx.

`enet_qos_rx_free_callback_t rxBuffFree`
Callback to free memory, must be provided for zero-copy Rx.

`struct __enet_qos_state`
#include <fsl_enet_qos.h> Defines the ENET state structure.

Note: The structure contains saved state for the instance. It could be stored in `enet_qos_handle_t`, but that's used only with the transactional API.

Public Members

`enet_qos_mii_mode_t miiMode`
MII mode.

`struct __enet_qos_buffer_struct`
#include <fsl_enet_qos.h> Defines the frame buffer structure.

Public Members

void *buffer

The buffer store the whole or partial frame.

uint16_t length

The byte length of this buffer.

struct _enet_qos_rx_frame_error

#include <fsl_enet_qos.h> Defines the Rx frame error structure.

Public Members

bool rxDstAddrFilterErr

Destination Address Filter Fail.

bool rxSrcAddrFilterErr

SA Address Filter Fail.

bool rxDribbleErr

Dribble error.

bool rxReceiveErr

Receive error.

bool rxOverflowErr

Receive over flow.

bool rxWatchDogErr

Watch dog timeout.

bool rxGaintPacketErr

Receive gaint packet.

bool rxCrcErr

Receive CRC error.

struct _enet_qos_rx_frame_attribute_struct

#include <fsl_enet_qos.h>

Public Members

bool isTsAvail

Rx frame timestamp is available or not.

enet_qos_ptp_time_t timestamp

The nanosecond part timestamp of this Rx frame.

struct _enet_qos_rx_frame_struct

#include <fsl_enet_qos.h> Defines the Rx frame data structure.

Public Members

enet_qos_buffer_struct_t *rxBuffArray

Rx frame buffer structure.

uint16_t totLen

Rx frame total length.

enet_qos_rx_frame_attribute_t rxAttribute

Rx frame attribute structure.

enet_qos_rx_frame_error_t rxFrameError

Rx frame error.

struct *_enet_qos_transfer_stats*

#include <fsl_enet_qos.h> Defines the ENET QOS transfer statistics structure.

Public Members

uint32_t statsRxFrameCount

Rx frame number.

uint32_t statsRxCrcErr

Rx frame number with CRC error.

uint32_t statsRxAlignErr

Rx frame number with alignment error.

uint32_t statsRxLengthErr

Rx frame length field doesn't equal to packet size.

uint32_t statsRxFifoOverflowErr

Rx FIFO overflow count.

uint32_t statsTxFrameCount

Tx frame number.

uint32_t statsTxFifoUnderRunErr

Tx FIFO underrun count.

2.7 FGPIO Driver

2.8 FlexCAN: Flex Controller Area Network Driver

2.9 FlexCAN Driver

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, 40000000UL);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.

`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->enablePretendedNetwork = 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_SetRxMbGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive message buffer global mask.

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Message Buffer Global Mask value.

void FLEXCAN_SetRxFifoGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive FIFO global mask.

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Fifo Global Mask value.

void FLEXCAN_SetRxIndividualMask(CAN_Type *base, uint8_t maskIdx, uint32_t mask)

Sets the FlexCAN receive individual mask.

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the FLEXCAN_Init(). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

- base – FlexCAN peripheral base address.
- maskIdx – The Index of individual Mask.
- mask – Rx Individual Mask value.

void FLEXCAN_SetTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

void FLEXCAN_SetRxMbConfig(CAN_Type *base, uint8_t mbIdx, const flexcan_rx_mb_config_t *pRxMbConfig, bool enable)

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer. User should invoke this API when CTRL2[RRS]=1. When CTRL2[RRS]=1, frame's

ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbEmpty, kFLEXCAN_RxMbFull or kFLEXCAN_RxMbOverrun. Message buffer will store the remote frame in the same fashion of a data frame. No automatic remote response frame will be generated. User need to setup another message buffer to respond remote request.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

```
static inline void FLEXCAN_SetMbID(CAN_Type *base, uint8_t mbIdx, uint32_t id)
```

Configures a FlexCAN Message Buffer identifier.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- id – CAN Message Buffer Identifier, should use FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

```
void FLEXCAN_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.

static inline uint32_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, uint32_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 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_EnableInterrupts(CAN_Type *base, uint32_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, uint32_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_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 *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_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_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.

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.

```
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_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_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.

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.

enum _flexcan_frame_format

FlexCAN frame format.

Values:

enumerator kFLEXCAN_FrameFormatStandard

Standard frame format attribute.

enumerator kFLEXCAN_FrameFormatExtend

Extend frame format attribute.

enum _flexcan_frame_type

FlexCAN frame type.

Values:

enumerator kFLEXCAN_FrameTypeData

Data frame type attribute.

enumerator kFLEXCAN_FrameTypeRemote

Remote frame type attribute.

enum _flexcan_clock_source

FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

Values:

enumerator kFLEXCAN_ClkSrcOsc

FlexCAN Protocol Engine clock from Oscillator.

enumerator kFLEXCAN_ClkSrcPeri

FlexCAN Protocol Engine clock from Peripheral Clock.

enumerator kFLEXCAN_ClkSrc0

FlexCAN Protocol Engine clock selected by user as SRC == 0.

enumerator kFLEXCAN_ClkSrc1

FlexCAN Protocol Engine clock selected by user as SRC == 1.

enum _flexcan_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_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.

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_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

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_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 occupies 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.

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_rx_fifo_filter_type* flexcan_rx_fifo_filter_type_t

FlexCAN Rx Fifo Filter type.

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 struct *_flexcan_frame* flexcan_frame_t

FlexCAN message frame structure.

`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 struct flexcan_rx_fifo_config flexcan_rx_fifo_config_t`
FlexCAN Legacy 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, uint32_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.

FLEXCAN_ID_STD(id)
FlexCAN frame length helper macro.

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.

`FLEXCAN_ERROR_AND_STATUS_INT_FLAG`

FlexCAN interrupt/status flag helper macro.

`FLEXCAN_PNWAKE_UP_FLAG`

`FLEXCAN_WAKE_UP_FLAG`

`FLEXCAN_MEMORY_ERROR_INT_FLAG`

`FLEXCAN_ENHANCED_RX_FIFO_INT_FLAG`

FlexCAN Enhanced Rx FIFO base address helper macro.

`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_frame`

`#include <fsl_flexcan.h>` FlexCAN message frame structure.

`struct _flexcan_timing_config`

`#include <fsl_flexcan.h>` FlexCAN protocol timing characteristic configuration structure.

Public Members

`uint32_t preDivider`

Classic CAN or CAN FD nominal phase bit rate prescaler.

`uint32_t rJumpwidth`

Classic CAN or CAN FD nominal phase Re-sync Jump Width.

`uint32_t phaseSeg1`

Classic CAN or CAN FD nominal phase Segment 1.

`uint32_t phaseSeg2`

Classic CAN or CAN FD nominal phase Segment 2.

uint32_t propSeg

Classic CAN or CAN FD nominal phase Propagation Segment.

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.

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 enableSupervisorMode

Enable or Disable Supervisor Mode, enable this mode will make registers allow only Supervisor access.

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_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_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_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_frame_t *volatile rxFifoFrameBuf
```

The buffer for received CAN data from Legacy 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 __unnamed16__
```

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 __unnamed18__
```

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 __unnamed20__
```

Public Members

struct __flexcan_frame

struct __flexcan_frame

struct ____unnamed22____

Public Members

uint32_t dataWord0

CAN Frame payload word0.

uint32_t dataWord1

CAN Frame payload word1.

struct ____unnamed24____

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 ____unnamed26____

Public Members

struct __flexcan_config

struct __flexcan_config

struct ____unnamed28____

Public Members

uint32_t baudRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

struct ____unnamed30____

Public Members

uint32_t bitRate

FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

2.10 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 flexcan_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.

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.

2.11 FlexIO: FlexIO Driver

2.12 FlexIO Driver

`void FLEXIO_GetDefaultConfig(flexio_config_t *userConfig)`

Gets the default configuration to configure the FlexIO module. The configuration can be used directly to call the `FLEXIO_Configure()`.

Example:

```
flexio_config_t config;
FLEXIO_GetDefaultConfig(&config);
```

Parameters

- `userConfig` – pointer to `flexio_config_t` structure

`void FLEXIO_Init(FLEXIO_Type *base, const flexio_config_t *userConfig)`

Configures the FlexIO with a FlexIO configuration. The configuration structure can be filled by the user or be set with default values by `FLEXIO_GetDefaultConfig()`.

Example

```
flexio_config_t config = {
    .enableFlexio = true,
    .enableInDoze = false,
    .enableInDebug = true,
    .enableFastAccess = false
};
FLEXIO_Configure(base, &config);
```

Parameters

- `base` – FlexIO peripheral base address
- `userConfig` – pointer to `flexio_config_t` structure

`void FLEXIO_Deinit(FLEXIO_Type *base)`

Gates the FlexIO clock. Call this API to stop the FlexIO clock.

Note: After calling this API, call the `FLEXIO_Init` to use the FlexIO module.

Parameters

- `base` – FlexIO peripheral base address

```
uint32_t FLEXIO_GetInstance(FLEXIO_Type *base)
```

Get instance number for FLEXIO module.

Parameters

- base – FLEXIO peripheral base address.

```
void FLEXIO_Reset(FLEXIO_Type *base)
```

Resets the FlexIO module.

Parameters

- base – FlexIO peripheral base address

```
static inline void FLEXIO_Enable(FLEXIO_Type *base, bool enable)
```

Enables the FlexIO module operation.

Parameters

- base – FlexIO peripheral base address
- enable – true to enable, false to disable.

```
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,
```

(continues on next page)

(continued from previous page)

```

.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 $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_DisableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the shifter error interrupt. The interrupt won't generate when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

static inline void FLEXIO_EnableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Enables the timer status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline void FLEXIO_DisableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)
Disables the timer status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

static inline uint32_t FLEXIO_GetShifterStatusFlags(FLEXIO_Type *base)
Gets the shifter status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter status flags

```
static inline void FLEXIO_ClearShifterStatusFlags(FLEXIO_Type *base, uint32_t mask)
```

Clears the shifter status flags.

Note: For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

```
static inline uint32_t FLEXIO_GetShifterErrorFlags(FLEXIO_Type *base)
```

Gets the shifter error flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter error flags

```
static inline void FLEXIO_ClearShifterErrorFlags(FLEXIO_Type *base, uint32_t mask)
```

Clears the shifter error flags.

Note: For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by $(1 \ll \text{shifter index})$

```
static inline uint32_t FLEXIO_GetTimerStatusFlags(FLEXIO_Type *base)
```

Gets the timer status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Timer status flags

```
static inline void FLEXIO_ClearTimerStatusFlags(FLEXIO_Type *base, uint32_t mask)
```

Clears the timer status flags.

Note: For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using $((1 \ll \text{timer index0}) | (1 \ll \text{timer index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by $(1 \ll \text{timer index})$

```
static inline void FLEXIO_EnableShifterStatusDMA(FLEXIO_Type *base, uint32_t mask, bool
                                                enable)
```

Enables/disables the shifter status DMA. The DMA request generates when the corresponding SSF is set.

Note: For multiple shifter status DMA enables, for example, calculate the mask by using $((1 \ll \text{shifter index0}) | (1 \ll \text{shifter index1}))$

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$
- enable – True to enable, false to disable.

```
uint32_t FLEXIO_GetShifterBufferAddress(FLEXIO_Type *base, flexio_shifter_buffer_type_t type,
                                       uint8_t index)
```

Gets the shifter buffer address for the DMA transfer usage.

Parameters

- base – FlexIO peripheral base address
- type – Shifter type of `flexio_shifter_buffer_type_t`
- index – Shifter index

Returns

Corresponding shifter buffer index

```
status_t FLEXIO_RegisterHandleIRQ(void *base, void *handle, flexio_isr_t isr)
```

Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- handle – Pointer to the handler for FlexIO simulated peripheral.
- isr – FlexIO simulated peripheral interrupt handler.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

```
status_t FLEXIO_UnregisterHandleIRQ(void *base)
```

Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

```
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.

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.

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 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.

`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.

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.

2.13 FlexIO eDMA I2S Driver


```
void FLEXIO_I2S_TransferTxCreateHandleEDMA(FLEXIO_I2S_Type *base,
                                           flexio_i2s_edma_handle_t *handle,
                                           flexio_i2s_edma_callback_t callback, void
                                           *userData, edma_handle_t *dmaHandle)
```

Initializes the FlexIO I2S eDMA handle.

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer.
- callback – FlexIO I2S eDMA callback function called while finished a block.
- userData – User parameter for callback.
- dmaHandle – eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

```
void FLEXIO_I2S_TransferRxCreateHandleEDMA(FLEXIO_I2S_Type *base,
                                           flexio_i2s_edma_handle_t *handle,
                                           flexio_i2s_edma_callback_t callback, void
                                           *userData, edma_handle_t *dmaHandle)
```

Initializes the FlexIO I2S Rx eDMA handle.

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer.
- callback – FlexIO I2S eDMA callback function called while finished a block.
- userData – User parameter for callback.
- dmaHandle – eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

```
void FLEXIO_I2S_TransferSetFormatEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                       *handle, flexio_i2s_format_t *format, uint32_t
                                       srcClock_Hz)
```

Configures the FlexIO I2S Tx audio format.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to format.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S eDMA handle pointer
- format – Pointer to FlexIO I2S audio data format structure.
- srcClock_Hz – FlexIO I2S clock source frequency in Hz, it should be 0 while in slave mode.

```
status_t FLEXIO_I2S_TransferSendEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                     *handle, flexio_i2s_transfer_t *xfer)
```

Performs a non-blocking FlexIO I2S transfer using DMA.

Note: This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetTransferStatus to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- kStatus__Success – Start a FlexIO I2S eDMA send successfully.
- kStatus__InvalidArgument – The input arguments is invalid.
- kStatus__TxBusy – FlexIO I2S is busy sending data.

```
status_t FLEXIO_I2S_TransferReceiveEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                         *handle, flexio_i2s_transfer_t *xfer)
```

Performs a non-blocking FlexIO I2S receive using eDMA.

Note: This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetReceiveRemainingBytes to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- kStatus__Success – Start a FlexIO I2S eDMA receive successfully.
- kStatus__InvalidArgument – The input arguments is invalid.
- kStatus__RxBusy – FlexIO I2S is busy receiving data.

```
void FLEXIO_I2S_TransferAbortSendEDMA(FLEXIO_I2S_Type *base, flexio_i2s_edma_handle_t
                                       *handle)
```

Aborts a FlexIO I2S transfer using eDMA.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.

```
void FLEXIO_I2S_TransferAbortReceiveEDMA(FLEXIO_I2S_Type *base,
                                          flexio_i2s_edma_handle_t *handle)
```

Aborts a FlexIO I2S receive using eDMA.

Parameters

- base – FlexIO I2S peripheral base address.

- handle – FlexIO I2S DMA handle pointer.

status_t FLEXIO_I2S_TransferGetSendCountEDMA(*FLEXIO_I2S_Type* *base,
flexio_i2s_edma_handle_t *handle, size_t
*count)

Gets the remaining bytes to be sent.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- count – Bytes sent.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

status_t FLEXIO_I2S_TransferGetReceiveCountEDMA(*FLEXIO_I2S_Type* *base,
flexio_i2s_edma_handle_t *handle, size_t
*count)

Get the remaining bytes to be received.

Parameters

- base – FlexIO I2S peripheral base address.
- handle – FlexIO I2S DMA handle pointer.
- count – Bytes received.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

FSL_FLEXIO_I2S_EDMA_DRIVER_VERSION

FlexIO I2S EDMA driver version 2.1.9.

typedef struct flexio_i2s_edma_handle flexio_i2s_edma_handle_t

typedef void (*flexio_i2s_edma_callback_t)(*FLEXIO_I2S_Type* *base, flexio_i2s_edma_handle_t
*handle, *status_t* status, void *userData)

FlexIO I2S eDMA transfer callback function for finish and error.

struct flexio_i2s_edma_handle

#include <fsl_flexio_i2s_edma.h> FlexIO I2S DMA transfer handle, users should not touch the
content of the handle.

Public Members

edma_handle_t *dmaHandle

DMA handler for FlexIO I2S send

uint8_t bytesPerFrame

Bytes in a frame

uint8_t nbytes

eDMA minor byte transfer count initially configured.

`uint32_t state`
Internal state for FlexIO I2S eDMA transfer

`flexio_i2s_edma_callback_t callback`
Callback for users while transfer finish or error occurred

`void *userData`
User callback parameter

`edma_tcd_t tcd[(4U) + 1U]`
TCD pool for eDMA transfer.

`flexio_i2s_transfer_t queue[(4U)]`
Transfer queue storing queued transfer.

`size_t transferSize[(4U)]`
Data bytes need to transfer

`volatile uint8_t queueUser`
Index for user to queue transfer.

`volatile uint8_t queueDriver`
Index for driver to get the transfer data and size

2.14 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

2.15 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

2.16 FlexIO I2C Master Driver

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.

2.17 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

2.18 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*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)*2=8. Example

```
FLEXIO_SPI_Type spiDev = {  
    .flexioBase = FLEXIO,  
    .SDOPinIndex = 0,  
    .SDIPinIndex = 1,  
    .SCKPinIndex = 2,  
    .CSnPinIndex = 3,  
    .shifterIndex = {0,1},  
    .timerIndex = {0}  
};  
flexio_spi_slave_config_t config = {  
    .enableSlave = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false,  
    .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,  
    .direction = kFLEXIO_SPI_MsbFirst,  
    .dataMode = kFLEXIO_SPI_8BitMode  
};  
FLEXIO_SPI_SlaveInit(&spiDev, &config);
```

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

void FLEXIO_SPI_SlaveDeinit(*FLEXIO_SPI_Type* *base)

Gates the FlexIO clock.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_SlaveGetDefaultConfig(*flexio_spi_slave_config_t* *slaveConfig)

Gets the default configuration to configure the FlexIO SPI slave. The configuration can be used directly for calling the FLEXIO_SPI_SlaveConfigure(). Example:

```
flexio_spi_slave_config_t slaveConfig;
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

uint32_t FLEXIO_SPI_GetStatusFlags(*FLEXIO_SPI_Type* *base)

Gets FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- kFLEXIO_SPI_TxEmptyFlag
- kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_ClearStatusFlags(*FLEXIO_SPI_Type* *base, uint32_t mask)

Clears FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – status flag The parameter can be any combination of the following values:
 - kFLEXIO_SPI_TxEmptyFlag
 - kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_EnableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Enables the FlexIO SPI interrupt.

This function enables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source. The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

```
void FLEXIO_SPI_DisableInterrupts(FLEXIO_SPI_Type *base, uint32_t mask)
```

Disables the FlexIO SPI interrupt.

This function disables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- mask – interrupt source The parameter can be any combination of the following values:
 - kFLEXIO_SPI_RxFullInterruptEnable
 - kFLEXIO_SPI_TxEmptyInterruptEnable

```
void FLEXIO_SPI_EnableDMA(FLEXIO_SPI_Type *base, uint32_t mask, bool enable)
```

Enables/disables the FlexIO SPI transmit DMA. This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO_SPI_TxEmptyFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- mask – SPI DMA source.
- enable – True means enable DMA, false means disable DMA.

```
static inline uint32_t FLEXIO_SPI_GetTxDataRegisterAddress(FLEXIO_SPI_Type *base,  
                                                         flexio_spi_shift_direction_t  
                                                         direction)
```

Gets the FlexIO SPI transmit data register address for MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI transmit data register address.

```
static inline uint32_t FLEXIO_SPI_GetRxDataRegisterAddress(FLEXIO_SPI_Type *base,  
                                                         flexio_spi_shift_direction_t  
                                                         direction)
```

Gets the FlexIO SPI receive data register address for the MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.

Returns

FlexIO SPI receive data register address.

```
static inline void FLEXIO_SPI_Enable(FLEXIO_SPI_Type *base, bool enable)
```

Enables/disables the FlexIO SPI module operation.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type*.
- enable – True to enable, false does not have any effect.


```
void FLEXIO_SPI_MasterSetBaudRate(FLEXIO_SPI_Type *base, uint32_t baudRate_Bps,  
                                uint32_t srcClockHz)
```

Sets baud rate for the FlexIO SPI transfer, which is only used for the master.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- baudRate_Bps – Baud Rate needed in Hz.
- srcClockHz – SPI source clock frequency in Hz.

```
static inline void FLEXIO_SPI_WriteData(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                       direction, uint32_t data)
```

Writes one byte of data, which is sent using the MSB method.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.
- data – 8/16/32 bit data.

```
static inline uint32_t FLEXIO_SPI_ReadData(FLEXIO_SPI_Type *base,  
                                           flexio_spi_shift_direction_t direction)
```

Reads 8 bit/16 bit data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.

Returns

8 bit/16 bit data received.

```
status_t FLEXIO_SPI_WriteBlocking(FLEXIO_SPI_Type *base, flexio_spi_shift_direction_t  
                                  direction, const uint8_t *buffer, size_t size)
```

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the *FLEXIO_SPI_Type* structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The data bytes to send.
- size – The number of data bytes to send.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

status_t FLEXIO_SPI_ReadBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction, *uint8_t* *buffer, *size_t* size)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The buffer to store the received bytes.
- size – The number of data bytes to be received.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

status_t FLEXIO_SPI_MasterTransferBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_transfer_t* *xfer)

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – pointer to FLEXIO_SPI_Type structure
- xfer – FlexIO SPI transfer structure, see flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

void FLEXIO_SPI_FlushShifters(*FLEXIO_SPI_Type* *base)

Flush tx/rx shifters.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

status_t FLEXIO_SPI_MasterTransferCreateHandle(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, *flexio_spi_master_transfer_callback_t* callback, *void* *userData)

Initializes the FlexIO SPI Master handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- callback – The callback function.

- `userData` – The parameter of the callback function.

Return values

- `kStatus_Success` – Successfully create the handle.
- `kStatus_OutOfRange` – The FlexIO type/handle/ISR table out of range.

`status_t` FLEXIO_SPI_MasterTransferNonBlocking(*FLEXIO_SPI_Type* *base,
flexio_spi_master_handle_t *handle,
flexio_spi_transfer_t *xfer)

Master transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- `base` – Pointer to the FLEXIO_SPI_Type structure.
- `handle` – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- `xfer` – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_FLEXIO_SPI_Busy` – SPI is not idle, is running another transfer.

`void` FLEXIO_SPI_MasterTransferAbort(*FLEXIO_SPI_Type* *base, flexio_spi_master_handle_t *handle)

Aborts the master data transfer, which used IRQ.

Parameters

- `base` – Pointer to the FLEXIO_SPI_Type structure.
- `handle` – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

`status_t` FLEXIO_SPI_MasterTransferGetCount(*FLEXIO_SPI_Type* *base,
flexio_spi_master_handle_t *handle, size_t *count)

Gets the data transfer status which used IRQ.

Parameters

- `base` – Pointer to the FLEXIO_SPI_Type structure.
- `handle` – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – count is Invalid.
- `kStatus_Success` – Successfully return the count.

`void` FLEXIO_SPI_MasterTransferHandleIRQ(`void` *spiType, `void` *spiHandle)

FlexIO SPI master IRQ handler function.

Parameters

- `spiType` – Pointer to the FLEXIO_SPI_Type structure.

- spiHandle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

```
status_t FLEXIO_SPI_SlaveTransferCreateHandle(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle,  
                                              flexio_spi_slave_transfer_callback_t callback,  
                                              void *userData)
```

Initializes the FlexIO SPI Slave handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus__Success – Successfully create the handle.
- kStatus__OutOfRange – The FlexIO type/handle/ISR table out of range.

```
status_t FLEXIO_SPI_SlaveTransferNonBlocking(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle,  
                                              flexio_spi_transfer_t *xfer)
```

Slave transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- base – Pointer to the FLEXIO_SPI_Type structure.
- xfer – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- kStatus__Success – Successfully start a transfer.
- kStatus__InvalidArgument – Input argument is invalid.
- kStatus__FLEXIO_SPI_Busy – SPI is not idle; it is running another transfer.

```
static inline void FLEXIO_SPI_SlaveTransferAbort(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle)
```

Aborts the slave data transfer which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

```
static inline status_t FLEXIO_SPI_SlaveTransferGetCount(FLEXIO_SPI_Type *base,  
                                              flexio_spi_slave_handle_t *handle,  
                                              size_t *count)
```

Gets the data transfer status which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

- `handle` – Pointer to the `flexio_spi_slave_handle_t` structure to store the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void FLEXIO_SPI_SlaveTransferHandleIRQ(void *spiType, void *spiHandle)`

FlexIO SPI slave IRQ handler function.

Parameters

- `spiType` – Pointer to the `FLEXIO_SPI_Type` structure.
- `spiHandle` – Pointer to the `flexio_spi_slave_handle_t` structure to store the transfer state.

`FSL_FLEXIO_SPI_DRIVER_VERSION`

FlexIO SPI driver version.

Error codes for the FlexIO SPI driver.

Values:

enumerator `kStatus_FLEXIO_SPI_Busy`

FlexIO SPI is busy.

enumerator `kStatus_FLEXIO_SPI_Idle`

SPI is idle

enumerator `kStatus_FLEXIO_SPI_Error`

FlexIO SPI error.

enumerator `kStatus_FLEXIO_SPI_Timeout`

FlexIO SPI timeout polling status flags.

enum `_flexio_spi_clock_phase`

FlexIO SPI clock phase configuration.

Values:

enumerator `kFLEXIO_SPI_ClockPhaseFirstEdge`

First edge on SPCK occurs at the middle of the first cycle of a data transfer.

enumerator `kFLEXIO_SPI_ClockPhaseSecondEdge`

First edge on SPCK occurs at the start of the first cycle of a data transfer.

enum `_flexio_spi_shift_direction`

FlexIO SPI data shifter direction options.

Values:

enumerator `kFLEXIO_SPI_MsbFirst`

Data transfers start with most significant bit.

enumerator `kFLEXIO_SPI_LsbFirst`

Data transfers start with least significant bit.

enum `_flexio_spi_data_bitcount_mode`

FlexIO SPI data length mode options.

Values:

enumerator kFLEXIO_SPI_8BitMode
8-bit data transmission mode.

enumerator kFLEXIO_SPI_16BitMode
16-bit data transmission mode.

enumerator kFLEXIO_SPI_32BitMode
32-bit data transmission mode.

enum _flexio_spi_interrupt_enable
Define FlexIO SPI interrupt mask.

Values:

enumerator kFLEXIO_SPI_TxEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_SPI_RxFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_spi_status_flags
Define FlexIO SPI status mask.

Values:

enumerator kFLEXIO_SPI_TxBufferEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_SPI_RxBufferFullFlag
Receive buffer full flag.

enum _flexio_spi_dma_enable
Define FlexIO SPI DMA mask.

Values:

enumerator kFLEXIO_SPI_TxDmaEnable
Tx DMA request source

enumerator kFLEXIO_SPI_RxDmaEnable
Rx DMA request source

enumerator kFLEXIO_SPI_DmaAllEnable
All DMA request source

enum _flexio_spi_transfer_flags
Define FlexIO SPI transfer flags.

Note: Use kFLEXIO_SPI_csContinuous and one of the other flags to OR together to form the transfer flag.

Values:

enumerator kFLEXIO_SPI_8bitMsb
FlexIO SPI 8-bit MSB first

enumerator kFLEXIO_SPI_8bitLsb
FlexIO SPI 8-bit LSB first

enumerator kFLEXIO_SPI_16bitMsb
FlexIO SPI 16-bit MSB first

```

enumerator kFLEXIO__SPI__16bitLsb
    FlexIO SPI 16-bit LSB first
enumerator kFLEXIO__SPI__32bitMsb
    FlexIO SPI 32-bit MSB first
enumerator kFLEXIO__SPI__32bitLsb
    FlexIO SPI 32-bit LSB first
enumerator kFLEXIO__SPI__csContinuous
    Enable the CS signal continuous mode
typedef enum _flexio_spi_clock_phase flexio_spi_clock_phase_t
    FlexIO SPI clock phase configuration.
typedef enum _flexio_spi_shift_direction flexio_spi_shift_direction_t
    FlexIO SPI data shifter direction options.
typedef enum _flexio_spi_data_bitcount_mode flexio_spi_data_bitcount_mode_t
    FlexIO SPI data length mode options.
typedef struct _flexio_spi_type FLEXIO__SPI__Type
    Define FlexIO SPI access structure typedef.
typedef struct _flexio_spi_master_config flexio_spi_master_config_t
    Define FlexIO SPI master configuration structure.
typedef struct _flexio_spi_slave_config flexio_spi_slave_config_t
    Define FlexIO SPI slave configuration structure.
typedef struct _flexio_spi_transfer flexio_spi_transfer_t
    Define FlexIO SPI transfer structure.
typedef struct _flexio_spi_master_handle flexio_spi_master_handle_t
    typedef for flexio_spi_master_handle_t in advance.
typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t
    Slave handle is the same with master handle.
typedef void (*flexio_spi_master_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_master_handle_t *handle, status_t status, void *userData)
    FlexIO SPI master callback for finished transmit.
typedef void (*flexio_spi_slave_transfer_callback_t)(FLEXIO_SPI_Type *base,
flexio_spi_slave_handle_t *handle, status_t status, void *userData)
    FlexIO SPI slave callback for finished transmit.
FLEXIO__SPI__DUMMYDATA
    FlexIO SPI dummy transfer data, the data is sent while txData is NULL.
SPI_RETRY_TIMES
    Retry times for waiting flag.
FLEXIO__SPI__XFER_DATA_FORMAT(flag)
    Get the transfer data format of width and bit order.
struct _flexio_spi_type
    #include <fsl_flexio_spi.h> Define FlexIO SPI access structure typedef.

```

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t SDOPinIndex

Pin select for data output. To set SDO pin in Hi-Z state, user needs to mux the pin as GPIO input and disable all pull up/down in application.

uint8_t SDIPinIndex

Pin select for data input.

uint8_t SCKPinIndex

Pin select for clock.

uint8_t CSnPinIndex

Pin select for enable.

uint8_t shifterIndex[2]

Shifter index used in FlexIO SPI.

uint8_t timerIndex[2]

Timer index used in FlexIO SPI.

struct _flexio_spi_master_config

#include <fsl_flexio_spi.h> Define FlexIO SPI master configuration structure.

Public Members

bool enableMaster

Enable/disable FlexIO SPI master after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct _flexio_spi_slave_config

#include <fsl_flexio_spi.h> Define FlexIO SPI slave configuration structure.

Public Members

bool enableSlave

Enable/disable FlexIO SPI slave after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct *_flexio_spi_transfer*

#include <fsl_flexio_spi.h> Define FlexIO SPI transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

size_t dataSize

Transfer bytes.

uint8_t flags

FlexIO SPI control flag, MSB first or LSB first.

struct *_flexio_spi_master_handle*

#include <fsl_flexio_spi.h> Define FlexIO SPI handle structure.

Public Members

const uint8_t *txData

Transfer buffer.

uint8_t *rxData

Receive buffer.

size_t transferSize

Total bytes to be transferred.

volatile size_t txRemainingBytes

Send data remaining in bytes.

volatile size_t rxRemainingBytes

Receive data remaining in bytes.

volatile uint32_t state

FlexIO SPI internal state.

uint8_t bytePerFrame

SPI mode, 2bytes or 1byte in a frame

flexio_spi_shift_direction_t direction

Shift direction.

flexio_spi_master_transfer_callback_t callback

FlexIO SPI callback.

void *userData

Callback parameter.

bool isCsContinuous

Is current transfer using CS continuous mode.

uint32_t timer1Cfg

TIMER1 TIMCFG regiser value backup.

2.19 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 ____unnamed106____

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.

2.20 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

bool enableAHBWriteIpTxFifo

Enable AHB bus write access to IP TX FIFO.

bool enableAHBWriteIpRxFifo

Enable AHB bus write access to IP RX FIFO.

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.

2.21 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_EDMAAnSize1Bytes(one byte).

status_t FLEXSPI_TransferEDMA(FLEXSPI_Type *base, *flexspi_edma_handle_t* *handle, *flexspi_transfer_t* *xfer)

Transfers FLEXSPI data using an eDMA non-blocking method.

This function writes/receives data to/from the FLEXSPI transmit/receive FIFO. This function is non-blocking.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to *flexspi_edma_handle_t* structure
- xfer – FLEXSPI transfer structure.

Return values

- kStatus_FLEXSPI_Busy – FLEXSPI is busy transfer.
- kStatus_InvalidArgument – The watermark configuration is invalid, the watermark should be power of 2 to do successfully EDMA transfer.
- kStatus_Success – FLEXSPI successfully start edma transfer.

void FLEXSPI_TransferAbortEDMA(FLEXSPI_Type *base, *flexspi_edma_handle_t* *handle)

Aborts the transfer data using eDMA.

This function aborts the transfer data using eDMA.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to *flexspi_edma_handle_t* structure

status_t FLEXSPI_TransferGetTransferCountEDMA(FLEXSPI_Type *base, *flexspi_edma_handle_t* *handle, *size_t* *count)

Gets the transferred counts of transfer.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to *flexspi_edma_handle_t* structure.
- count – Bytes transfer.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`FSL_FLEXSPI_EDMA_DRIVER_VERSION`

FLEXSPI EDMA driver version.

FLEXSPI EDMA driver.

`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

`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

`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

2.22 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`.

2.23 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.

uint8_t masterDynamicAddress

Main master dynamic address configuration.

uint32_t slowClock_Hz

Slow clock frequency for time control.

uint32_t maxWriteLength

Maximum write length.

uint32_t maxReadLength

Maximum read length.

bool enableSlave

Whether to enable slave.

bool isHotJoin

Whether to enable slave hotjoin before enable slave.

uint8_t staticAddr

Static address.

uint16_t vendorID

Device vendor ID(manufacture ID).

bool enableRandomPart

Whether to generate random part number, if using random part number, the part-Number variable setting is meaningless.

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.

2.24 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_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.

`void I3C_SlaveRequestEvent(I3C_Type *base, i3c_slave_event_t event)`
I3C slave request event.

Parameters

- `base` – The I3C peripheral base address.
- `event` – I3C slave event of type `i3c_slave_event_t`

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`
Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`
Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_MasterTransferCreateHandle(I3C_Type *base, i3c_master_handle_t *handle, const i3c_master_transfer_callback_t *callback, void *userData)`

Creates a new handle for the I3C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C master driver handle.

- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_MasterTransferNonBlocking(I3C_Type *base, i3c_master_handle_t *handle, i3c_master_transfer_t *transfer)`

Performs a non-blocking transaction on the I2C/I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.
- `transfer` – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

`status_t I3C_MasterTransferGetCount(I3C_Type *base, i3c_master_handle_t *handle, size_t *count)`

Returns number of bytes transferred so far.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.
- `count` – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_Success` –
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`void I3C_MasterTransferAbort(I3C_Type *base, i3c_master_handle_t *handle)`

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I3C peripheral's IRQ priority.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.

`void I3C_MasterTransferHandleIRQ(I3C_Type *base, void *intHandle)`

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The I3C peripheral base address.

- intHandle – Pointer to the I3C master driver handle.

enum _i3c_master_flags

I3C master peripheral flags.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterBetweenFlag

Between messages/DAA's flag

enumerator kI3C_MasterNackDetectFlag

NACK detected flag

enumerator kI3C_MasterSlaveStartFlag

Slave request start flag

enumerator kI3C_MasterControlDoneFlag

Master request complete flag

enumerator kI3C_MasterCompleteFlag

Transfer complete flag

enumerator kI3C_MasterRxReadyFlag

Rx data ready in Rx buffer flag

enumerator kI3C_MasterTxReadyFlag

Tx buffer ready for Tx data flag

enumerator kI3C_MasterArbitrationWonFlag

Header address won arbitration flag

enumerator kI3C_MasterErrorFlag

Error occurred flag

enumerator kI3C_MasterSlave2MasterFlag

Switch from slave to master flag

enumerator kI3C_MasterClearFlags

enum _i3c_master_error_flags

I3C master error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterErrorNackFlag
Slave NACKed the last address

enumerator kI3C_MasterErrorWriteAbortFlag
Slave NACKed the write data

enumerator kI3C_MasterErrorTermFlag
Master terminates slave read

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.

uint32_t slowClock_Hz

Slow clock frequency.

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.

2.25 I3C Master DMA Driver

```
void I3C_MasterTransferCreateHandleEDMA(I3C_Type *base, i3c_master_edma_handle_t
                                         *handle, const i3c_master_edma_callback_t
                                         *callback, void *userData, edma_handle_t
                                         *rxDmaHandle, edma_handle_t *txDmaHandle)
```

Create a new handle for the I3C 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 `I3C_MasterTransferAbortDMA()` API shall be called.

For devices where the I3C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – 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.
- *rxDmaHandle* – Handle for the DMA receive channel. Created by the user prior to calling this function.
- *txDmaHandle* – Handle for the DMA transmit channel. Created by the user prior to calling this function.

```
status_t I3C_MasterTransferEDMA(I3C_Type *base, i3c_master_edma_handle_t *handle,
                                i3c_master_transfer_t *transfer)
```

Performs a non-blocking DMA-based transaction on the I3C bus.

The callback specified when the *handle* was created is invoked when the transaction has completed.

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 another DMA transaction is already in progress.

`status_t I3C_MasterTransferGetCountEDMA(I3C_Type *base, i3c_master_edma_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 DMA transaction currently in progress.

`void I3C_MasterTransferAbortEDMA(I3C_Type *base, i3c_master_edma_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 DMA peripheral's IRQ priority.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – Pointer to the I3C master driver handle.

`void I3C_MasterTransferEDMAHandleIRQ(I3C_Type *base, void *i3cHandle)`

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.
- *i3cHandle* – Pointer to the I3C master DMA driver handle.


```
typedef struct _i3c_master_edma_handle i3c_master_edma_handle_t  
typedef struct _i3c_master_edma_callback i3c_master_edma_callback_t  
    i3c master callback functions.  
struct _i3c_master_edma_callback  
    #include <fsl_i3c_edma.h> i3c master callback functions.
```

Public Members

```
void (*slave2Master)(I3C_Type *base, void *userData)  
    Target asks for controller request.  
void (*ibiCallback)(I3C_Type *base, i3c_master_edma_handle_t *handle, i3c_ibi_type_t  
ibiType, i3c_ibi_state_t ibiState)  
    IBI event callback.  
void (*transferComplete)(I3C_Type *base, i3c_master_edma_handle_t *handle, status_t  
status, void *userData)  
    Transfer complete callback.
```

```
struct _i3c_master_edma_handle  
    #include <fsl_i3c_edma.h> Driver handle for master EDMA APIs.
```

Note: The contents of this structure are private and subject to change.

Public Members

```
I3C_Type *base  
    I3C base pointer.  
uint8_t state  
    Transfer state machine current state.  
uint32_t transferCount  
    Indicates progress of the transfer  
uint8_t subaddressBuffer[4]  
    Saving subaddress command.  
uint8_t subaddressCount  
    Saving command count.  
i3c_master_transfer_t transfer  
    Copy of the current transfer info.  
i3c_master_edma_callback_t callback  
    Callback function pointer.  
void *userData  
    Application data passed to callback.  
edma_handle_t *rxDmaHandle  
    Handle for receive DMA channel.  
edma_handle_t *txDmaHandle  
    Handle for transmit DMA channel.
```

`bool ibiFlag`
IBIWON flag.

`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.

`status_t result`
Transfer result.

2.26 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.

void I3C_SlaveRequestEvent(I3C_Type *base, *i3c_slave_event_t* event)

I3C slave request event.

Parameters

- `base` – The I3C peripheral base address.

- `event` – I3C slave event of type `i3c_slave_event_t`

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_SlaveTransferCreateHandle(I3C_Type *base, i3c_slave_handle_t *handle, i3c_slave_transfer_callback_t callback, void *userData)`

Creates a new handle for the I3C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_SlaveTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C slave driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_SlaveTransferNonBlocking(I3C_Type *base, i3c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling `I2C_SlaveInit()` and `I3C_SlaveTransferCreateHandle()` to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to `I3C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i3c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI3C_SlaveTransmitEvent` and `kI3C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In

addition, the `kI3C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i3c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI3C_SlaveAllEvents` to enable all events.

Return values

- `kStatus__Success` – Slave transfers were successfully started.
- `kStatus__I3C_Busy` – Slave transfers have already been started on this handle.

`status_t I3C_SlaveTransferGetCount(I3C_Type *base, i3c_slave_handle_t *handle, size_t *count)`

Gets the slave transfer status during a non-blocking transfer.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure.
- `count` – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- `kStatus__Success` –
- `kStatus__NoTransferInProgress` –

`void I3C_SlaveTransferAbort(I3C_Type *base, i3c_slave_handle_t *handle)`

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`void I3C_SlaveTransferHandleIRQ(I3C_Type *base, void *intHandle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The I3C peripheral base address.
- `intHandle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`void I3C_SlaveRequestIBIWithData(I3C_Type *base, uint8_t *data, size_t dataSize)`
I3C slave request IBI event with data payload(mandatory and extended).

Parameters

- `base` – The I3C peripheral base address.
- `data` – Pointer to IBI data to be sent in the request.
- `dataSize` – IBI data size.

`void I3C_SlaveRequestIBIWithSingleData(I3C_Type *base, uint8_t data, size_t dataSize)`
I3C slave request IBI event with single data.

Deprecated:

Do not use this function. It has been superseded by `I3C_SlaveRequestIBIWithData`.

Parameters

- `base` – The I3C peripheral base address.
- `data` – IBI data to be sent in the request.
- `dataSize` – IBI data size.

`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 ENTDA mode

enumerator kI3C_SlaveBusDAAFlag

I3C bus is in ENTDA mode

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.

bool isHotJoin

Whether to enable slave hotjoin before enable slave.

uint8_t staticAddr

Static address.

uint16_t vendorID

Device vendor ID(manufacture ID).

bool enableRandomPart

Whether to generate random part number, if using random part number, the part-Number variable setting is meaningless.

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 I3C_SlaveCompletionEvent.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct *_i3c_slave_handle*

#include <fsl_i3c.h> I3C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

i3c_slave_transfer_t transfer

I3C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

i3c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

size_t txFifoSize

Tx Fifo size

2.27 I3C Slave DMA Driver

```
void I3C_SlaveTransferCreateHandleEDMA(I3C_Type *base, i3c_slave_edma_handle_t *handle,
                                       i3c_slave_edma_callback_t callback, void *userData,
                                       edma_handle_t *rxDmaHandle, edma_handle_t
                                       *txDmaHandle)
```

Create a new handle for the I3C slave 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 I3C_SlaveTransferAbortDMA() API shall be called.

For devices where the I3C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- base – The I3C peripheral base address.
- handle – 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.
- `rxDmaHandle` – Handle for the DMA receive channel. Created by the user prior to calling this function.
- `txDmaHandle` – Handle for the DMA transmit channel. Created by the user prior to calling this function.

```
status_t I3C_SlaveTransferEDMA(I3C_Type *base, i3c_slave_edma_handle_t *handle,  
                              i3c_slave_edma_transfer_t *transfer, uint32_t eventMask)
```

Prepares for a non-blocking DMA-based transaction on the I3C bus.

The API will do DMA configuration according to the input transfer descriptor, and the data will be transferred when there's bus master requesting transfer from/to this slave. So the timing of call to this API need be aligned with master application to ensure the transfer is executed as expected. Callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C slave driver handle.
- `transfer` – The pointer to the transfer descriptor.
- `eventMask` – Bit mask formed by OR'ing together `i3c_slave_transfer_event_t` enumerators to specify which events to send to the callback. The transmit and receive events is not allowed to be enabled.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.
- `kStatus_Fail` – The transaction can't be set.

```
void I3C_SlaveTransferAbortEDMA(I3C_Type *base, i3c_slave_edma_handle_t *handle)
```

Abort a slave edma non-blocking transfer in a early time.

Parameters

- `base` – I3C peripheral base address
- `handle` – pointer to `i3c_slave_edma_handle_t` structure

```
void I3C_SlaveTransferEDMAHandleIRQ(I3C_Type *base, void *i3CHandle)
```

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.
- `i3CHandle` – Pointer to the I3C slave DMA driver handle.

```
typedef struct _i3c_slave_edma_handle i3c_slave_edma_handle_t
```

```
typedef struct _i3c_slave_edma_transfer i3c_slave_edma_transfer_t  
I3C slave transfer structure.
```



```
typedef void (*i3c_slave_edma_callback_t)(I3C_Type *base, i3c_slave_edma_transfer_t *transfer, void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave DMA transfer API.

Param base

Base address for the I3C instance on which the event occurred.

Param handle

Pointer to slave DMA transfer handle.

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 _i3c_slave_edma_transfer
```

#include <fsl_i3c_edma.h> I3C slave transfer structure.

Public Members

uint32_t event

Reason the callback is being invoked.

uint8_t *txData

Transfer buffer

size_t txDataSize

Transfer size

uint8_t *rxData

Transfer buffer

size_t rxDataSize

Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI3C_SlaveCompletionEvent.

```
struct _i3c_slave_edma_handle
```

#include <fsl_i3c_edma.h> I3C slave edma handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

I3C_Type *base

I3C base pointer.

i3c_slave_edma_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.

`i3c_slave_edma_callback_t` `callback`
Callback function called at transfer event.

`edma_handle_t` `*rxDmaHandle`
Handle for receive DMA channel.

`edma_handle_t` `*txDmaHandle`
Handle for transmit DMA channel.

`void` `*userData`
Callback parameter passed to callback.

2.28 Iomuxc_driver

```
static inline void IOMUXC_SetPinMux(uint32_t muxRegister, uint32_t muxMode, uint32_t
                                     inputRegister, uint32_t inputDaisy, uint32_t
                                     configRegister, uint32_t inputOnfield)
```

Sets the IOMUXC pin mux mode.

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
- `inputOn` – The software input on

```
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.

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 1.0.0.

IOMUXC_PAD_DAP_TDI__JTAG_MUX_TDI

IOMUXC_PAD_DAP_TDI__MQS2_LEFT

IOMUXC_PAD_DAP_TDI__CAN2_TX

IOMUXC_PAD_DAP_TDI__FLEXIO2_FLEXIO30

IOMUXC_PAD_DAP_TDI__GPIO3_IO28

IOMUXC_PAD_DAP_TDI__LPUART5_RX

IOMUXC_PAD_DAP_TMS_SWDIO__JTAG_MUX_TMS

IOMUXC_PAD_DAP_TMS_SWDIO__FLEXIO2_FLEXIO31

IOMUXC_PAD_DAP_TMS_SWDIO__GPIO3_IO29

IOMUXC_PAD_DAP_TMS_SWDIO__LPUART5_RTS_B

IOMUXC_PAD_DAP_TCLK_SWCLK__JTAG_MUX_TCK

IOMUXC_PAD_DAP_TCLK_SWCLK__FLEXIO1_FLEXIO30

IOMUXC_PAD_DAP_TCLK_SWCLK__GPIO3_IO30

IOMUXC_PAD_DAP_TCLK_SWCLK__LPUART5_CTS_B

IOMUXC_PAD_DAP_TDO_TRACESWO__JTAG_MUX_TDO

IOMUXC_PAD_DAP_TDO_TRACESWO__MQS2_RIGHT

IOMUXC_PAD_DAP_TDO_TRACESWO__CAN2_RX

IOMUXC_PAD_DAP_TDO_TRACESWO__FLEXIO1_FLEXIO31

IOMUXC_PAD_DAP_TDO_TRACESWO__GPIO3_IO31

IOMUXC_PAD_DAP_TDO_TRACESWO__LPUART5_TX

IOMUXC_PAD_GPIO_IO00__GPIO2_IO00

IOMUXC_PAD_GPIO_IO00__LPI2C3_SDA

IOMUXC_PAD_GPIO_IO00__MEDIAMIX_CAM_CLK

IOMUXC_PAD_GPIO_IO00__MEDIAMIX_DISP_CLK

IOMUXC_PAD_GPIO_IO00__LPSPI6_PCS0

IOMUXC_PAD_GPIO_IO00__LPUART5_TX

IOMUXC_PAD_GPIO_IO00__LPI2C5_SDA

IOMUXC_PAD_GPIO_IO00__FLEXIO1_FLEXIO00

IOMUXC_PAD_GPIO_IO01__GPIO2_IO01

IOMUXC_PAD_GPIO_IO01__LPI2C3_SCL

IOMUXC_PAD_GPIO_IO01__MEDIAMIX_CAM_DATA00

IOMUXC_PAD_GPIO_IO01__MEDIAMIX_DISP_DE
IOMUXC_PAD_GPIO_IO01__LPSPI6_SIN
IOMUXC_PAD_GPIO_IO01__LPUART5_RX
IOMUXC_PAD_GPIO_IO01__LPI2C5_SCL
IOMUXC_PAD_GPIO_IO01__FLEXIO1_FLEXIO01
IOMUXC_PAD_GPIO_IO02__GPIO2_IO02
IOMUXC_PAD_GPIO_IO02__LPI2C4_SDA
IOMUXC_PAD_GPIO_IO02__MEDIAMIX_CAM_VSYNC
IOMUXC_PAD_GPIO_IO02__MEDIAMIX_DISP_VSYNC
IOMUXC_PAD_GPIO_IO02__LPSPI6_SOUT
IOMUXC_PAD_GPIO_IO02__LPUART5_CTS_B
IOMUXC_PAD_GPIO_IO02__LPI2C6_SDA
IOMUXC_PAD_GPIO_IO02__FLEXIO1_FLEXIO02
IOMUXC_PAD_GPIO_IO03__GPIO2_IO03
IOMUXC_PAD_GPIO_IO03__LPI2C4_SCL
IOMUXC_PAD_GPIO_IO03__MEDIAMIX_CAM_HSYNC
IOMUXC_PAD_GPIO_IO03__MEDIAMIX_DISP_HSYNC
IOMUXC_PAD_GPIO_IO03__LPSPI6_SCK
IOMUXC_PAD_GPIO_IO03__LPUART5_RTS_B
IOMUXC_PAD_GPIO_IO03__LPI2C6_SCL
IOMUXC_PAD_GPIO_IO03__FLEXIO1_FLEXIO03
IOMUXC_PAD_GPIO_IO04__GPIO2_IO04
IOMUXC_PAD_GPIO_IO04__TPM3_CH0
IOMUXC_PAD_GPIO_IO04__PDM_CLK
IOMUXC_PAD_GPIO_IO04__MEDIAMIX_DISP_DATA00
IOMUXC_PAD_GPIO_IO04__LPSPI7_PCS0
IOMUXC_PAD_GPIO_IO04__LPUART6_TX
IOMUXC_PAD_GPIO_IO04__LPI2C6_SDA
IOMUXC_PAD_GPIO_IO04__FLEXIO1_FLEXIO04
IOMUXC_PAD_GPIO_IO05__GPIO2_IO05
IOMUXC_PAD_GPIO_IO05__TPM4_CH0
IOMUXC_PAD_GPIO_IO05__PDM_BIT_STREAM00
IOMUXC_PAD_GPIO_IO05__MEDIAMIX_DISP_DATA01

IOMUXC_PAD_GPIO_IO05__LPSPI7_SIN
IOMUXC_PAD_GPIO_IO05__LPUART6_RX
IOMUXC_PAD_GPIO_IO05__LPI2C6_SCL
IOMUXC_PAD_GPIO_IO05__FLEXIO1_FLEXIO05
IOMUXC_PAD_GPIO_IO06__GPIO2_IO06
IOMUXC_PAD_GPIO_IO06__TPM5_CH0
IOMUXC_PAD_GPIO_IO06__PDM_BIT_STREAM01
IOMUXC_PAD_GPIO_IO06__MEDIAMIX_DISP_DATA02
IOMUXC_PAD_GPIO_IO06__LPSPI7_SOUT
IOMUXC_PAD_GPIO_IO06__LPUART6_CTS_B
IOMUXC_PAD_GPIO_IO06__LPI2C7_SDA
IOMUXC_PAD_GPIO_IO06__FLEXIO1_FLEXIO06
IOMUXC_PAD_GPIO_IO07__GPIO2_IO07
IOMUXC_PAD_GPIO_IO07__LPSPI3_PCS1
IOMUXC_PAD_GPIO_IO07__MEDIAMIX_CAM_DATA01
IOMUXC_PAD_GPIO_IO07__MEDIAMIX_DISP_DATA03
IOMUXC_PAD_GPIO_IO07__LPSPI7_SCK
IOMUXC_PAD_GPIO_IO07__LPUART6_RTS_B
IOMUXC_PAD_GPIO_IO07__LPI2C7_SCL
IOMUXC_PAD_GPIO_IO07__FLEXIO1_FLEXIO07
IOMUXC_PAD_GPIO_IO08__GPIO2_IO08
IOMUXC_PAD_GPIO_IO08__LPSPI3_PCS0
IOMUXC_PAD_GPIO_IO08__MEDIAMIX_CAM_DATA02
IOMUXC_PAD_GPIO_IO08__MEDIAMIX_DISP_DATA04
IOMUXC_PAD_GPIO_IO08__TPM6_CH0
IOMUXC_PAD_GPIO_IO08__LPUART7_TX
IOMUXC_PAD_GPIO_IO08__LPI2C7_SDA
IOMUXC_PAD_GPIO_IO08__FLEXIO1_FLEXIO08
IOMUXC_PAD_GPIO_IO09__GPIO2_IO09
IOMUXC_PAD_GPIO_IO09__LPSPI3_SIN
IOMUXC_PAD_GPIO_IO09__MEDIAMIX_CAM_DATA03
IOMUXC_PAD_GPIO_IO09__MEDIAMIX_DISP_DATA05
IOMUXC_PAD_GPIO_IO09__TPM3_EXTCLK

IOMUXC_PAD_GPIO_IO09__LPUART7_RX
IOMUXC_PAD_GPIO_IO09__LPI2C7_SCL
IOMUXC_PAD_GPIO_IO09__FLEXIO1_FLEXIO09
IOMUXC_PAD_GPIO_IO10__GPIO2_IO10
IOMUXC_PAD_GPIO_IO10__LPSPI3_SOUT
IOMUXC_PAD_GPIO_IO10__MEDIAMIX_CAM_DATA04
IOMUXC_PAD_GPIO_IO10__MEDIAMIX_DISP_DATA06
IOMUXC_PAD_GPIO_IO10__TPM4_EXTCLK
IOMUXC_PAD_GPIO_IO10__LPUART7_CTS_B
IOMUXC_PAD_GPIO_IO10__LPI2C8_SDA
IOMUXC_PAD_GPIO_IO10__FLEXIO1_FLEXIO10
IOMUXC_PAD_GPIO_IO11__GPIO2_IO11
IOMUXC_PAD_GPIO_IO11__LPSPI3_SCK
IOMUXC_PAD_GPIO_IO11__MEDIAMIX_CAM_DATA05
IOMUXC_PAD_GPIO_IO11__MEDIAMIX_DISP_DATA07
IOMUXC_PAD_GPIO_IO11__TPM5_EXTCLK
IOMUXC_PAD_GPIO_IO11__LPUART7_RTS_B
IOMUXC_PAD_GPIO_IO11__LPI2C8_SCL
IOMUXC_PAD_GPIO_IO11__FLEXIO1_FLEXIO11
IOMUXC_PAD_GPIO_IO12__GPIO2_IO12
IOMUXC_PAD_GPIO_IO12__TPM3_CH2
IOMUXC_PAD_GPIO_IO12__PDM_BIT_STREAM02
IOMUXC_PAD_GPIO_IO12__MEDIAMIX_DISP_DATA08
IOMUXC_PAD_GPIO_IO12__LPSPI8_PCS0
IOMUXC_PAD_GPIO_IO12__LPUART8_TX
IOMUXC_PAD_GPIO_IO12__LPI2C8_SDA
IOMUXC_PAD_GPIO_IO12__SAI3_RX_SYNC
IOMUXC_PAD_GPIO_IO13__GPIO2_IO13
IOMUXC_PAD_GPIO_IO13__TPM4_CH2
IOMUXC_PAD_GPIO_IO13__PDM_BIT_STREAM03
IOMUXC_PAD_GPIO_IO13__MEDIAMIX_DISP_DATA09
IOMUXC_PAD_GPIO_IO13__LPSPI8_SIN
IOMUXC_PAD_GPIO_IO13__LPUART8_RX

IOMUXC_PAD_GPIO_IO13__LPI2C8_SCL
IOMUXC_PAD_GPIO_IO13__FLEXIO1_FLEXIO13
IOMUXC_PAD_GPIO_IO14__GPIO2_IO14
IOMUXC_PAD_GPIO_IO14__LPUART3_TX
IOMUXC_PAD_GPIO_IO14__MEDIAMIX_CAM_DATA06
IOMUXC_PAD_GPIO_IO14__MEDIAMIX_DISP_DATA10
IOMUXC_PAD_GPIO_IO14__LPSPI8_SOUT
IOMUXC_PAD_GPIO_IO14__LPUART8_CTS_B
IOMUXC_PAD_GPIO_IO14__LPUART4_TX
IOMUXC_PAD_GPIO_IO14__FLEXIO1_FLEXIO14
IOMUXC_PAD_GPIO_IO15__GPIO2_IO15
IOMUXC_PAD_GPIO_IO15__LPUART3_RX
IOMUXC_PAD_GPIO_IO15__MEDIAMIX_CAM_DATA07
IOMUXC_PAD_GPIO_IO15__MEDIAMIX_DISP_DATA11
IOMUXC_PAD_GPIO_IO15__LPSPI8_SCK
IOMUXC_PAD_GPIO_IO15__LPUART8_RTS_B
IOMUXC_PAD_GPIO_IO15__LPUART4_RX
IOMUXC_PAD_GPIO_IO15__FLEXIO1_FLEXIO15
IOMUXC_PAD_GPIO_IO16__GPIO2_IO16
IOMUXC_PAD_GPIO_IO16__SAI3_TX_BCLK
IOMUXC_PAD_GPIO_IO16__PDM_BIT_STREAM02
IOMUXC_PAD_GPIO_IO16__MEDIAMIX_DISP_DATA12
IOMUXC_PAD_GPIO_IO16__LPUART3_CTS_B
IOMUXC_PAD_GPIO_IO16__LPSPI4_PCS2
IOMUXC_PAD_GPIO_IO16__LPUART4_CTS_B
IOMUXC_PAD_GPIO_IO16__FLEXIO1_FLEXIO16
IOMUXC_PAD_GPIO_IO17__GPIO2_IO17
IOMUXC_PAD_GPIO_IO17__SAI3_MCLK
IOMUXC_PAD_GPIO_IO17__MEDIAMIX_CAM_DATA08
IOMUXC_PAD_GPIO_IO17__MEDIAMIX_DISP_DATA13
IOMUXC_PAD_GPIO_IO17__LPUART3_RTS_B
IOMUXC_PAD_GPIO_IO17__LPSPI4_PCS1
IOMUXC_PAD_GPIO_IO17__LPUART4_RTS_B

IOMUXC_PAD_GPIO_IO17___FLEXIO1_FLEXIO17
IOMUXC_PAD_GPIO_IO18___GPIO2_IO18
IOMUXC_PAD_GPIO_IO18___SAI3_RX_BCLK
IOMUXC_PAD_GPIO_IO18___MEDIAMIX_CAM_DATA09
IOMUXC_PAD_GPIO_IO18___MEDIAMIX_DISP_DATA14
IOMUXC_PAD_GPIO_IO18___LPSPI5_PCS0
IOMUXC_PAD_GPIO_IO18___LPSPI4_PCS0
IOMUXC_PAD_GPIO_IO18___TPM5_CH2
IOMUXC_PAD_GPIO_IO18___FLEXIO1_FLEXIO18
IOMUXC_PAD_GPIO_IO19___GPIO2_IO19
IOMUXC_PAD_GPIO_IO19___SAI3_RX_SYNC
IOMUXC_PAD_GPIO_IO19___PDM_BIT_STREAM03
IOMUXC_PAD_GPIO_IO19___MEDIAMIX_DISP_DATA15
IOMUXC_PAD_GPIO_IO19___LPSPI5_SIN
IOMUXC_PAD_GPIO_IO19___LPSPI4_SIN
IOMUXC_PAD_GPIO_IO19___TPM6_CH2
IOMUXC_PAD_GPIO_IO19___SAI3_TX_DATA00
IOMUXC_PAD_GPIO_IO20___GPIO2_IO20
IOMUXC_PAD_GPIO_IO20___SAI3_RX_DATA00
IOMUXC_PAD_GPIO_IO20___PDM_BIT_STREAM00
IOMUXC_PAD_GPIO_IO20___MEDIAMIX_DISP_DATA16
IOMUXC_PAD_GPIO_IO20___LPSPI5_SOUT
IOMUXC_PAD_GPIO_IO20___LPSPI4_SOUT
IOMUXC_PAD_GPIO_IO20___TPM3_CH1
IOMUXC_PAD_GPIO_IO20___FLEXIO1_FLEXIO20
IOMUXC_PAD_GPIO_IO21___GPIO2_IO21
IOMUXC_PAD_GPIO_IO21___SAI3_TX_DATA00
IOMUXC_PAD_GPIO_IO21___PDM_CLK
IOMUXC_PAD_GPIO_IO21___MEDIAMIX_DISP_DATA17
IOMUXC_PAD_GPIO_IO21___LPSPI5_SCK
IOMUXC_PAD_GPIO_IO21___LPSPI4_SCK
IOMUXC_PAD_GPIO_IO21___TPM4_CH1
IOMUXC_PAD_GPIO_IO21___SAI3_RX_BCLK

IOMUXC_PAD_GPIO_IO22___GPIO2_IO22
IOMUXC_PAD_GPIO_IO22___USDHC3_CLK
IOMUXC_PAD_GPIO_IO22___SPDIF_IN
IOMUXC_PAD_GPIO_IO22___MEDIAMIX_DISP_DATA18
IOMUXC_PAD_GPIO_IO22___TPM5_CH1
IOMUXC_PAD_GPIO_IO22___TPM6_EXTCLK
IOMUXC_PAD_GPIO_IO22___LPI2C5_SDA
IOMUXC_PAD_GPIO_IO22___FLEXIO1_FLEXIO22
IOMUXC_PAD_GPIO_IO23___GPIO2_IO23
IOMUXC_PAD_GPIO_IO23___USDHC3_CMD
IOMUXC_PAD_GPIO_IO23___SPDIF_OUT
IOMUXC_PAD_GPIO_IO23___MEDIAMIX_DISP_DATA19
IOMUXC_PAD_GPIO_IO23___TPM6_CH1
IOMUXC_PAD_GPIO_IO23___LPI2C5_SCL
IOMUXC_PAD_GPIO_IO23___FLEXIO1_FLEXIO23
IOMUXC_PAD_GPIO_IO24___GPIO2_IO24
IOMUXC_PAD_GPIO_IO24___USDHC3_DATA0
IOMUXC_PAD_GPIO_IO24___MEDIAMIX_DISP_DATA20
IOMUXC_PAD_GPIO_IO24___TPM3_CH3
IOMUXC_PAD_GPIO_IO24___JTAG_MUX_TDO
IOMUXC_PAD_GPIO_IO24___LPSPI6_PCS1
IOMUXC_PAD_GPIO_IO24___FLEXIO1_FLEXIO24
IOMUXC_PAD_GPIO_IO25___GPIO2_IO25
IOMUXC_PAD_GPIO_IO25___USDHC3_DATA1
IOMUXC_PAD_GPIO_IO25___CAN2_TX
IOMUXC_PAD_GPIO_IO25___MEDIAMIX_DISP_DATA21
IOMUXC_PAD_GPIO_IO25___TPM4_CH3
IOMUXC_PAD_GPIO_IO25___JTAG_MUX_TCK
IOMUXC_PAD_GPIO_IO25___LPSPI7_PCS1
IOMUXC_PAD_GPIO_IO25___FLEXIO1_FLEXIO25
IOMUXC_PAD_GPIO_IO26___GPIO2_IO26
IOMUXC_PAD_GPIO_IO26___USDHC3_DATA2
IOMUXC_PAD_GPIO_IO26___PDM_BIT_STREAM01

IOMUXC_PAD_GPIO_IO26___MEDIAMIX_DISP_DATA22
IOMUXC_PAD_GPIO_IO26___TPM5_CH3
IOMUXC_PAD_GPIO_IO26___JTAG_MUX_TDI
IOMUXC_PAD_GPIO_IO26___LPSPI8_PCS1
IOMUXC_PAD_GPIO_IO26___SAI3_TX_SYNC
IOMUXC_PAD_GPIO_IO27___GPIO2_IO27
IOMUXC_PAD_GPIO_IO27___USDHC3_DATA3
IOMUXC_PAD_GPIO_IO27___CAN2_RX
IOMUXC_PAD_GPIO_IO27___MEDIAMIX_DISP_DATA23
IOMUXC_PAD_GPIO_IO27___TPM6_CH3
IOMUXC_PAD_GPIO_IO27___JTAG_MUX_TMS
IOMUXC_PAD_GPIO_IO27___LPSPI5_PCS1
IOMUXC_PAD_GPIO_IO27___FLEXIO1_FLEXIO27
IOMUXC_PAD_GPIO_IO28___GPIO2_IO28
IOMUXC_PAD_GPIO_IO28___LPI2C3_SDA
IOMUXC_PAD_GPIO_IO28___FLEXIO1_FLEXIO28
IOMUXC_PAD_GPIO_IO29___GPIO2_IO29
IOMUXC_PAD_GPIO_IO29___LPI2C3_SCL
IOMUXC_PAD_GPIO_IO29___FLEXIO1_FLEXIO29
IOMUXC_PAD_CCM_CLKO1___CCMSRCGPCMIX_CLKO1
IOMUXC_PAD_CCM_CLKO1___FLEXIO1_FLEXIO26
IOMUXC_PAD_CCM_CLKO1___GPIO3_IO26
IOMUXC_PAD_CCM_CLKO2___GPIO3_IO27
IOMUXC_PAD_CCM_CLKO2___CCMSRCGPCMIX_CLKO2
IOMUXC_PAD_CCM_CLKO2___FLEXIO1_FLEXIO27
IOMUXC_PAD_CCM_CLKO3___CCMSRCGPCMIX_CLKO3
IOMUXC_PAD_CCM_CLKO3___FLEXIO2_FLEXIO28
IOMUXC_PAD_CCM_CLKO3___GPIO4_IO28
IOMUXC_PAD_CCM_CLKO4___CCMSRCGPCMIX_CLKO4
IOMUXC_PAD_CCM_CLKO4___FLEXIO2_FLEXIO29
IOMUXC_PAD_CCM_CLKO4___GPIO4_IO29
IOMUXC_PAD_ENET1_MDC___ENET_QOS_MDC
IOMUXC_PAD_ENET1_MDC___LPUART3_DCB_B

IOMUXC_PAD_ENET1_MDC__I3C2_SCL
IOMUXC_PAD_ENET1_MDC__HSIOMIX_OTG_ID1
IOMUXC_PAD_ENET1_MDC__FLEXIO2_FLEXIO00
IOMUXC_PAD_ENET1_MDC__GPIO4_IO00
IOMUXC_PAD_ENET1_MDIO__ENET_QOS_MDIO
IOMUXC_PAD_ENET1_MDIO__LPUART3_RIN_B
IOMUXC_PAD_ENET1_MDIO__I3C2_SDA
IOMUXC_PAD_ENET1_MDIO__HSIOMIX_OTG_PWR1
IOMUXC_PAD_ENET1_MDIO__FLEXIO2_FLEXIO01
IOMUXC_PAD_ENET1_MDIO__GPIO4_IO01
IOMUXC_PAD_ENET1_TD3__ENET_QOS_RGMII_TD3
IOMUXC_PAD_ENET1_TD3__CAN2_TX
IOMUXC_PAD_ENET1_TD3__HSIOMIX_OTG_ID2
IOMUXC_PAD_ENET1_TD3__FLEXIO2_FLEXIO02
IOMUXC_PAD_ENET1_TD3__GPIO4_IO02
IOMUXC_PAD_ENET1_TD2__ENET_QOS_RGMII_TD2
IOMUXC_PAD_ENET1_TD2__CCM_ENET_QOS_CLOCK_GENERATE_REF_CLK
IOMUXC_PAD_ENET1_TD2__CAN2_RX
IOMUXC_PAD_ENET1_TD2__HSIOMIX_OTG_OC2
IOMUXC_PAD_ENET1_TD2__FLEXIO2_FLEXIO03
IOMUXC_PAD_ENET1_TD2__GPIO4_IO03
IOMUXC_PAD_ENET1_TD1__ENET_QOS_RGMII_TD1
IOMUXC_PAD_ENET1_TD1__LPUART3_RTS_B
IOMUXC_PAD_ENET1_TD1__I3C2_PUR
IOMUXC_PAD_ENET1_TD1__HSIOMIX_OTG_OC1
IOMUXC_PAD_ENET1_TD1__FLEXIO2_FLEXIO04
IOMUXC_PAD_ENET1_TD1__GPIO4_IO04
IOMUXC_PAD_ENET1_TD1__I3C2_PUR_B
IOMUXC_PAD_ENET1_TD0__ENET_QOS_RGMII_TD0
IOMUXC_PAD_ENET1_TD0__LPUART3_TX
IOMUXC_PAD_ENET1_TD0__FLEXIO2_FLEXIO05
IOMUXC_PAD_ENET1_TD0__GPIO4_IO05
IOMUXC_PAD_ENET1_TX_CTL__ENET_QOS_RGMII_TX_CTL

IOMUXC_PAD_ENET1_TX_CTL__LPUART3_DTR_B
IOMUXC_PAD_ENET1_TX_CTL__FLEXIO2_FLEXIO06
IOMUXC_PAD_ENET1_TX_CTL__GPIO4_IO06
IOMUXC_PAD_ENET1_TXC__CCM_ENET_QOS_CLOCK_GENERATE_TX_CLK
IOMUXC_PAD_ENET1_TXC__ENET_QOS_TX_ER
IOMUXC_PAD_ENET1_TXC__FLEXIO2_FLEXIO07
IOMUXC_PAD_ENET1_TXC__GPIO4_IO07
IOMUXC_PAD_ENET1_RX_CTL__ENET_QOS_RGMII_RX_CTL
IOMUXC_PAD_ENET1_RX_CTL__LPUART3_DSR_B
IOMUXC_PAD_ENET1_RX_CTL__HSIOMIX_OTG_PWR2
IOMUXC_PAD_ENET1_RX_CTL__FLEXIO2_FLEXIO08
IOMUXC_PAD_ENET1_RX_CTL__GPIO4_IO08
IOMUXC_PAD_ENET1_RXC__CCM_ENET_QOS_CLOCK_GENERATE_RX_CLK
IOMUXC_PAD_ENET1_RXC__ENET_QOS_RX_ER
IOMUXC_PAD_ENET1_RXC__FLEXIO2_FLEXIO09
IOMUXC_PAD_ENET1_RXC__GPIO4_IO09
IOMUXC_PAD_ENET1_RD0__ENET_QOS_RGMII_RD0
IOMUXC_PAD_ENET1_RD0__LPUART3_RX
IOMUXC_PAD_ENET1_RD0__FLEXIO2_FLEXIO10
IOMUXC_PAD_ENET1_RD0__GPIO4_IO10
IOMUXC_PAD_ENET1_RD1__ENET_QOS_RGMII_RD1
IOMUXC_PAD_ENET1_RD1__LPUART3_CTS_B
IOMUXC_PAD_ENET1_RD1__LPTMR2_ALT1
IOMUXC_PAD_ENET1_RD1__FLEXIO2_FLEXIO11
IOMUXC_PAD_ENET1_RD1__GPIO4_IO11
IOMUXC_PAD_ENET1_RD2__ENET_QOS_RGMII_RD2
IOMUXC_PAD_ENET1_RD2__LPTMR2_ALT2
IOMUXC_PAD_ENET1_RD2__FLEXIO2_FLEXIO12
IOMUXC_PAD_ENET1_RD2__GPIO4_IO12
IOMUXC_PAD_ENET1_RD3__ENET_QOS_RGMII_RD3
IOMUXC_PAD_ENET1_RD3__LPTMR2_ALT3
IOMUXC_PAD_ENET1_RD3__FLEXIO2_FLEXIO13
IOMUXC_PAD_ENET1_RD3__GPIO4_IO13

IOMUXC_PAD_ENET2_MDC__ENET1_MDC
IOMUXC_PAD_ENET2_MDC__LPUART4_DCB_B
IOMUXC_PAD_ENET2_MDC__SAI2_RX_SYNC
IOMUXC_PAD_ENET2_MDC__FLEXIO2_FLEXIO14
IOMUXC_PAD_ENET2_MDC__GPIO4_IO14
IOMUXC_PAD_ENET2_MDIO__ENET1_MDIO
IOMUXC_PAD_ENET2_MDIO__LPUART4_RIN_B
IOMUXC_PAD_ENET2_MDIO__SAI2_RX_BCLK
IOMUXC_PAD_ENET2_MDIO__FLEXIO2_FLEXIO15
IOMUXC_PAD_ENET2_MDIO__GPIO4_IO15
IOMUXC_PAD_ENET2_TD3__SAI2_RX_DATA00
IOMUXC_PAD_ENET2_TD3__FLEXIO2_FLEXIO16
IOMUXC_PAD_ENET2_TD3__GPIO4_IO16
IOMUXC_PAD_ENET2_TD3__ENET1_RGMII_TD3
IOMUXC_PAD_ENET2_TD2__ENET1_RGMII_TD2
IOMUXC_PAD_ENET2_TD2__ENET1_TX_CLK
IOMUXC_PAD_ENET2_TD2__SAI2_RX_DATA01
IOMUXC_PAD_ENET2_TD2__FLEXIO2_FLEXIO17
IOMUXC_PAD_ENET2_TD2__GPIO4_IO17
IOMUXC_PAD_ENET2_TD1__ENET1_RGMII_TD1
IOMUXC_PAD_ENET2_TD1__LPUART4_RTS_B
IOMUXC_PAD_ENET2_TD1__SAI2_RX_DATA02
IOMUXC_PAD_ENET2_TD1__FLEXIO2_FLEXIO18
IOMUXC_PAD_ENET2_TD1__GPIO4_IO18
IOMUXC_PAD_ENET2_TD0__ENET1_RGMII_TD0
IOMUXC_PAD_ENET2_TD0__LPUART4_TX
IOMUXC_PAD_ENET2_TD0__SAI2_RX_DATA03
IOMUXC_PAD_ENET2_TD0__FLEXIO2_FLEXIO19
IOMUXC_PAD_ENET2_TD0__GPIO4_IO19
IOMUXC_PAD_ENET2_TX_CTL__ENET1_RGMII_TX_CTL
IOMUXC_PAD_ENET2_TX_CTL__LPUART4_DTR_B
IOMUXC_PAD_ENET2_TX_CTL__SAI2_TX_SYNC
IOMUXC_PAD_ENET2_TX_CTL__FLEXIO2_FLEXIO20

IOMUXC_PAD_ENET2_TX_CTL__GPIO4_IO20
IOMUXC_PAD_ENET2_TXC__ENET1_RGMII_TXC
IOMUXC_PAD_ENET2_TXC__ENET1_TX_ER
IOMUXC_PAD_ENET2_TXC__SAI2_TX_BCLK
IOMUXC_PAD_ENET2_TXC__FLEXIO2_FLEXIO21
IOMUXC_PAD_ENET2_TXC__GPIO4_IO21
IOMUXC_PAD_ENET2_RX_CTL__ENET1_RGMII_RX_CTL
IOMUXC_PAD_ENET2_RX_CTL__LPUART4_DSR_B
IOMUXC_PAD_ENET2_RX_CTL__SAI2_TX_DATA00
IOMUXC_PAD_ENET2_RX_CTL__FLEXIO2_FLEXIO22
IOMUXC_PAD_ENET2_RX_CTL__GPIO4_IO22
IOMUXC_PAD_ENET2_RXC__ENET1_RGMII_RXC
IOMUXC_PAD_ENET2_RXC__ENET1_RX_ER
IOMUXC_PAD_ENET2_RXC__SAI2_TX_DATA01
IOMUXC_PAD_ENET2_RXC__FLEXIO2_FLEXIO23
IOMUXC_PAD_ENET2_RXC__GPIO4_IO23
IOMUXC_PAD_ENET2_RD0__ENET1_RGMII_RD0
IOMUXC_PAD_ENET2_RD0__LPUART4_RX
IOMUXC_PAD_ENET2_RD0__SAI2_TX_DATA02
IOMUXC_PAD_ENET2_RD0__FLEXIO2_FLEXIO24
IOMUXC_PAD_ENET2_RD0__GPIO4_IO24
IOMUXC_PAD_ENET2_RD1__ENET1_RGMII_RD1
IOMUXC_PAD_ENET2_RD1__SPDIF_IN
IOMUXC_PAD_ENET2_RD1__SAI2_TX_DATA03
IOMUXC_PAD_ENET2_RD1__FLEXIO2_FLEXIO25
IOMUXC_PAD_ENET2_RD1__GPIO4_IO25
IOMUXC_PAD_ENET2_RD2__ENET1_RGMII_RD2
IOMUXC_PAD_ENET2_RD2__LPUART4_CTS_B
IOMUXC_PAD_ENET2_RD2__SAI2_MCLK
IOMUXC_PAD_ENET2_RD2__MQS2_RIGHT
IOMUXC_PAD_ENET2_RD2__FLEXIO2_FLEXIO26
IOMUXC_PAD_ENET2_RD2__GPIO4_IO26
IOMUXC_PAD_ENET2_RD3__ENET1_RGMII_RD3

IOMUXC_PAD_ENET2_RD3__SPDIF_OUT
IOMUXC_PAD_ENET2_RD3__SPDIF_IN
IOMUXC_PAD_ENET2_RD3__MQS2_LEFT
IOMUXC_PAD_ENET2_RD3__FLEXIO2_FLEXIO27
IOMUXC_PAD_ENET2_RD3__GPIO4_IO27
IOMUXC_PAD_SD1_CLK__FLEXIO1_FLEXIO08
IOMUXC_PAD_SD1_CLK__GPIO3_IO08
IOMUXC_PAD_SD1_CLK__USDHC1_CLK
IOMUXC_PAD_SD1_CMD__USDHC1_CMD
IOMUXC_PAD_SD1_CMD__FLEXIO1_FLEXIO09
IOMUXC_PAD_SD1_CMD__GPIO3_IO09
IOMUXC_PAD_SD1_DATA0__USDHC1_DATA0
IOMUXC_PAD_SD1_DATA0__FLEXIO1_FLEXIO10
IOMUXC_PAD_SD1_DATA0__GPIO3_IO10
IOMUXC_PAD_SD1_DATA1__USDHC1_DATA1
IOMUXC_PAD_SD1_DATA1__FLEXIO1_FLEXIO11
IOMUXC_PAD_SD1_DATA1__GPIO3_IO11
IOMUXC_PAD_SD1_DATA2__USDHC1_DATA2
IOMUXC_PAD_SD1_DATA2__FLEXIO1_FLEXIO12
IOMUXC_PAD_SD1_DATA2__GPIO3_IO12
IOMUXC_PAD_SD1_DATA3__USDHC1_DATA3
IOMUXC_PAD_SD1_DATA3__FLEXSPI1_A_SS1_B
IOMUXC_PAD_SD1_DATA3__FLEXIO1_FLEXIO13
IOMUXC_PAD_SD1_DATA3__GPIO3_IO13
IOMUXC_PAD_SD1_DATA4__USDHC1_DATA4
IOMUXC_PAD_SD1_DATA4__FLEXSPI1_A_DATA04
IOMUXC_PAD_SD1_DATA4__FLEXIO1_FLEXIO14
IOMUXC_PAD_SD1_DATA4__GPIO3_IO14
IOMUXC_PAD_SD1_DATA5__USDHC1_DATA5
IOMUXC_PAD_SD1_DATA5__FLEXSPI1_A_DATA05
IOMUXC_PAD_SD1_DATA5__USDHC1_RESET_B
IOMUXC_PAD_SD1_DATA5__FLEXIO1_FLEXIO15
IOMUXC_PAD_SD1_DATA5__GPIO3_IO15

IOMUXC_PAD_SD1_DATA6___USDHC1_DATA6
IOMUXC_PAD_SD1_DATA6___FLEXSPI1_A_DATA06
IOMUXC_PAD_SD1_DATA6___USDHC1_CD_B
IOMUXC_PAD_SD1_DATA6___FLEXIO1_FLEXIO16
IOMUXC_PAD_SD1_DATA6___GPIO3_IO16
IOMUXC_PAD_SD1_DATA7___USDHC1_DATA7
IOMUXC_PAD_SD1_DATA7___FLEXSPI1_A_DATA07
IOMUXC_PAD_SD1_DATA7___USDHC1_WP
IOMUXC_PAD_SD1_DATA7___FLEXIO1_FLEXIO17
IOMUXC_PAD_SD1_DATA7___GPIO3_IO17
IOMUXC_PAD_SD1_STROBE___USDHC1_STROBE
IOMUXC_PAD_SD1_STROBE___FLEXSPI1_A_DQS
IOMUXC_PAD_SD1_STROBE___FLEXIO1_FLEXIO18
IOMUXC_PAD_SD1_STROBE___GPIO3_IO18
IOMUXC_PAD_SD2_VSELECT___USDHC2_VSELECT
IOMUXC_PAD_SD2_VSELECT___USDHC2_WP
IOMUXC_PAD_SD2_VSELECT___LPTMR2_ALT3
IOMUXC_PAD_SD2_VSELECT___FLEXIO1_FLEXIO19
IOMUXC_PAD_SD2_VSELECT___GPIO3_IO19
IOMUXC_PAD_SD2_VSELECT___CCMSRCGPCMIX_EXT_CLK1
IOMUXC_PAD_SD3_CLK___USDHC3_CLK
IOMUXC_PAD_SD3_CLK___FLEXSPI1_A_SCLK
IOMUXC_PAD_SD3_CLK___FLEXIO1_FLEXIO20
IOMUXC_PAD_SD3_CLK___GPIO3_IO20
IOMUXC_PAD_SD3_CMD___USDHC3_CMD
IOMUXC_PAD_SD3_CMD___FLEXSPI1_A_SS0_B
IOMUXC_PAD_SD3_CMD___FLEXIO1_FLEXIO21
IOMUXC_PAD_SD3_CMD___GPIO3_IO21
IOMUXC_PAD_SD3_DATA0___USDHC3_DATA0
IOMUXC_PAD_SD3_DATA0___FLEXSPI1_A_DATA00
IOMUXC_PAD_SD3_DATA0___FLEXIO1_FLEXIO22
IOMUXC_PAD_SD3_DATA0___GPIO3_IO22
IOMUXC_PAD_SD3_DATA1___USDHC3_DATA1

IOMUXC_PAD_SD3_DATA1__FLEXSPI1_A_DATA01
IOMUXC_PAD_SD3_DATA1__FLEXIO1_FLEXIO23
IOMUXC_PAD_SD3_DATA1__GPIO3_IO23
IOMUXC_PAD_SD3_DATA2__USDHC3_DATA2
IOMUXC_PAD_SD3_DATA2__FLEXSPI1_A_DATA02
IOMUXC_PAD_SD3_DATA2__FLEXIO1_FLEXIO24
IOMUXC_PAD_SD3_DATA2__GPIO3_IO24
IOMUXC_PAD_SD3_DATA3__USDHC3_DATA3
IOMUXC_PAD_SD3_DATA3__FLEXSPI1_A_DATA03
IOMUXC_PAD_SD3_DATA3__FLEXIO1_FLEXIO25
IOMUXC_PAD_SD3_DATA3__GPIO3_IO25
IOMUXC_PAD_SD2_CD_B__USDHC2_CD_B
IOMUXC_PAD_SD2_CD_B__ENET_QOS_1588_EVENT0_IN
IOMUXC_PAD_SD2_CD_B__I3C2_SCL
IOMUXC_PAD_SD2_CD_B__FLEXIO1_FLEXIO00
IOMUXC_PAD_SD2_CD_B__GPIO3_IO00
IOMUXC_PAD_SD2_CLK__USDHC2_CLK
IOMUXC_PAD_SD2_CLK__ENET_QOS_1588_EVENT0_OUT
IOMUXC_PAD_SD2_CLK__I3C2_SDA
IOMUXC_PAD_SD2_CLK__FLEXIO1_FLEXIO01
IOMUXC_PAD_SD2_CLK__GPIO3_IO01
IOMUXC_PAD_SD2_CLK__CCMSRCGPCMIX_OBSERVE0
IOMUXC_PAD_SD2_CMD__USDHC2_CMD
IOMUXC_PAD_SD2_CMD__ENET1_1588_EVENT0_IN
IOMUXC_PAD_SD2_CMD__I3C2_PUR
IOMUXC_PAD_SD2_CMD__I3C2_PUR_B
IOMUXC_PAD_SD2_CMD__FLEXIO1_FLEXIO02
IOMUXC_PAD_SD2_CMD__GPIO3_IO02
IOMUXC_PAD_SD2_CMD__CCMSRCGPCMIX_OBSERVE1
IOMUXC_PAD_SD2_DATA0__USDHC2_DATA0
IOMUXC_PAD_SD2_DATA0__ENET1_1588_EVENT0_OUT
IOMUXC_PAD_SD2_DATA0__CAN2_TX
IOMUXC_PAD_SD2_DATA0__FLEXIO1_FLEXIO03

IOMUXC_PAD_SD2_DATA0___GPIO3_IO03
IOMUXC_PAD_SD2_DATA0___CCMSRCGPCMIX_OBSERVE2
IOMUXC_PAD_SD2_DATA1___USDHC2_DATA1
IOMUXC_PAD_SD2_DATA1___ENET1_1588_EVENT1_IN
IOMUXC_PAD_SD2_DATA1___CAN2_RX
IOMUXC_PAD_SD2_DATA1___FLEXIO1_FLEXIO04
IOMUXC_PAD_SD2_DATA1___GPIO3_IO04
IOMUXC_PAD_SD2_DATA2___USDHC2_DATA2
IOMUXC_PAD_SD2_DATA2___ENET1_1588_EVENT1_OUT
IOMUXC_PAD_SD2_DATA2___MQS2_RIGHT
IOMUXC_PAD_SD2_DATA2___FLEXIO1_FLEXIO05
IOMUXC_PAD_SD2_DATA2___GPIO3_IO05
IOMUXC_PAD_SD2_DATA3___USDHC2_DATA3
IOMUXC_PAD_SD2_DATA3___LPTMR2_ALT1
IOMUXC_PAD_SD2_DATA3___MQS2_LEFT
IOMUXC_PAD_SD2_DATA3___FLEXIO1_FLEXIO06
IOMUXC_PAD_SD2_DATA3___GPIO3_IO06
IOMUXC_PAD_SD2_RESET_B___USDHC2_RESET_B
IOMUXC_PAD_SD2_RESET_B___LPTMR2_ALT2
IOMUXC_PAD_SD2_RESET_B___FLEXIO1_FLEXIO07
IOMUXC_PAD_SD2_RESET_B___GPIO3_IO07
IOMUXC_PAD_SD2_RESET_B___CCMSRCGPCMIX_SYSTEM_RESET
IOMUXC_PAD_I2C1_SCL___LPI2C1_SCL
IOMUXC_PAD_I2C1_SCL___I3C1_SCL
IOMUXC_PAD_I2C1_SCL___LPUART1_DCB_B
IOMUXC_PAD_I2C1_SCL___TPM2_CH0
IOMUXC_PAD_I2C1_SCL___GPIO1_IO00
IOMUXC_PAD_I2C1_SDA___LPI2C1_SDA
IOMUXC_PAD_I2C1_SDA___I3C1_SDA
IOMUXC_PAD_I2C1_SDA___LPUART1_RIN_B
IOMUXC_PAD_I2C1_SDA___TPM2_CH1
IOMUXC_PAD_I2C1_SDA___GPIO1_IO01
IOMUXC_PAD_I2C2_SCL___LPI2C2_SCL

IOMUXC_PAD_I2C2_SCL___I3C1_PUR
IOMUXC_PAD_I2C2_SCL___LPUART2_DCB_B
IOMUXC_PAD_I2C2_SCL___TPM2_CH2
IOMUXC_PAD_I2C2_SCL___SAI1_RX_SYNC
IOMUXC_PAD_I2C2_SCL___GPIO1_IO02
IOMUXC_PAD_I2C2_SCL___I3C1_PUR_B
IOMUXC_PAD_I2C2_SDA___LPI2C2_SDA
IOMUXC_PAD_I2C2_SDA___LPUART2_RIN_B
IOMUXC_PAD_I2C2_SDA___TPM2_CH3
IOMUXC_PAD_I2C2_SDA___SAI1_RX_BCLK
IOMUXC_PAD_I2C2_SDA___GPIO1_IO03
IOMUXC_PAD_UART1_RXD___LPUART1_RX
IOMUXC_PAD_UART1_RXD___S400_UART_RX
IOMUXC_PAD_UART1_RXD___LPSPI2_SIN
IOMUXC_PAD_UART1_RXD___TPM1_CH0
IOMUXC_PAD_UART1_RXD___GPIO1_IO04
IOMUXC_PAD_UART1_TXD___LPUART1_TX
IOMUXC_PAD_UART1_TXD___S400_UART_TX
IOMUXC_PAD_UART1_TXD___LPSPI2_PCS0
IOMUXC_PAD_UART1_TXD___TPM1_CH1
IOMUXC_PAD_UART1_TXD___GPIO1_IO05
IOMUXC_PAD_UART2_RXD___LPUART2_RX
IOMUXC_PAD_UART2_RXD___LPUART1_CTS_B
IOMUXC_PAD_UART2_RXD___LPSPI2_SOUT
IOMUXC_PAD_UART2_RXD___TPM1_CH2
IOMUXC_PAD_UART2_RXD___SAI1_MCLK
IOMUXC_PAD_UART2_RXD___GPIO1_IO06
IOMUXC_PAD_UART2_TXD___LPUART2_TX
IOMUXC_PAD_UART2_TXD___LPUART1_RTS_B
IOMUXC_PAD_UART2_TXD___LPSPI2_SCK
IOMUXC_PAD_UART2_TXD___TPM1_CH3
IOMUXC_PAD_UART2_TXD___GPIO1_IO07
IOMUXC_PAD_PDM_CLK___PDM_CLK

IOMUXC_PAD_PDM_CLK__MQS1_LEFT
IOMUXC_PAD_PDM_CLK__LPTMR1_ALT1
IOMUXC_PAD_PDM_CLK__GPIO1_IO08
IOMUXC_PAD_PDM_CLK__CAN1_TX
IOMUXC_PAD_PDM_BIT_STREAM0__PDM_BIT_STREAM00
IOMUXC_PAD_PDM_BIT_STREAM0__MQS1_RIGHT
IOMUXC_PAD_PDM_BIT_STREAM0__LPSPI1_PCS1
IOMUXC_PAD_PDM_BIT_STREAM0__TPM1_EXTCLK
IOMUXC_PAD_PDM_BIT_STREAM0__LPTMR1_ALT2
IOMUXC_PAD_PDM_BIT_STREAM0__GPIO1_IO09
IOMUXC_PAD_PDM_BIT_STREAM0__CAN1_RX
IOMUXC_PAD_PDM_BIT_STREAM1__PDM_BIT_STREAM01
IOMUXC_PAD_PDM_BIT_STREAM1__NMI_GLUE_NMI
IOMUXC_PAD_PDM_BIT_STREAM1__LPSPI2_PCS1
IOMUXC_PAD_PDM_BIT_STREAM1__TPM2_EXTCLK
IOMUXC_PAD_PDM_BIT_STREAM1__LPTMR1_ALT3
IOMUXC_PAD_PDM_BIT_STREAM1__GPIO1_IO10
IOMUXC_PAD_PDM_BIT_STREAM1__CCMSRCGPCMIX_EXT_CLK1
IOMUXC_PAD_SAI1_TXFS__SAI1_TX_SYNC
IOMUXC_PAD_SAI1_TXFS__SAI1_TX_DATA01
IOMUXC_PAD_SAI1_TXFS__LPSPI1_PCS0
IOMUXC_PAD_SAI1_TXFS__LPUART2_DTR_B
IOMUXC_PAD_SAI1_TXFS__MQS1_LEFT
IOMUXC_PAD_SAI1_TXFS__GPIO1_IO11
IOMUXC_PAD_SAI1_TXC__SAI1_TX_BCLK
IOMUXC_PAD_SAI1_TXC__LPUART2_CTS_B
IOMUXC_PAD_SAI1_TXC__LPSPI1_SIN
IOMUXC_PAD_SAI1_TXC__LPUART1_DSR_B
IOMUXC_PAD_SAI1_TXC__CAN1_RX
IOMUXC_PAD_SAI1_TXC__GPIO1_IO12
IOMUXC_PAD_SAI1_TXD0__SAI1_TX_DATA00
IOMUXC_PAD_SAI1_TXD0__LPUART2_RTS_B
IOMUXC_PAD_SAI1_TXD0__LPSPI1_SCK

IOMUXC_PAD_SAI1_TXD0__LPUART1_DTR_B
IOMUXC_PAD_SAI1_TXD0__CAN1_TX
IOMUXC_PAD_SAI1_TXD0__GPIO1_IO13
IOMUXC_PAD_SAI1_RXD0__SAI1_RX_DATA00
IOMUXC_PAD_SAI1_RXD0__SAI1_MCLK
IOMUXC_PAD_SAI1_RXD0__LPSPI1_SOUT
IOMUXC_PAD_SAI1_RXD0__LPUART2_DSR_B
IOMUXC_PAD_SAI1_RXD0__MQS1_RIGHT
IOMUXC_PAD_SAI1_RXD0__GPIO1_IO14
IOMUXC_PAD_WDOG_ANY__WDOG1_WDOG_ANY
IOMUXC_PAD_WDOG_ANY__GPIO1_IO15
IOMUXC_PAD_MUX_MODE_MASK
IOMUXC_PAD_MUX_MODE_SHIFT
IOMUXC_PAD_MUX_MODE(x)
IOMUXC_PAD_SION_MASK
IOMUXC_PAD_SION_SHIFT
IOMUXC_PAD_SION(x)
IOMUXC_PAD_DSE_MASK
IOMUXC_PAD_DSE_SHIFT
IOMUXC_PAD_DSE(x)
IOMUXC_PAD_FSEL1_MASK
IOMUXC_PAD_FSEL1_SHIFT
IOMUXC_PAD_FSEL1(x)
IOMUXC_PAD_PU_MASK
IOMUXC_PAD_PU_SHIFT
IOMUXC_PAD_PU(x)
IOMUXC_PAD_PD_MASK
IOMUXC_PAD_PD_SHIFT
IOMUXC_PAD_PD(x)
IOMUXC_PAD_OD_MASK
IOMUXC_PAD_OD_SHIFT
IOMUXC_PAD_OD(x)
IOMUXC_PAD_HYS_MASK

IOMUXC_PAD_HYS_SHIFT
IOMUXC_PAD_HYS(x)
IOMUXC_PAD_APC_MASK
IOMUXC_PAD_APC_SHIFT
IOMUXC_PAD_APC(x)
FSL_COMPONENT_ID

2.29 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.

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).

enum *_isi_roi_index*
ISI roi index number.
Values:

enumerator ISI_ROI_INDEX_0
ISI ROI index 0

enumerator ISI_ROI_INDEX_1
ISI ROI index 1

enumerator ISI_ROI_INDEX_2
ISI ROI index 2

enumerator ISI_ROI_INDEX_3
ISI ROI index 3

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_regain_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.

```
typedef enum _isi_roi_index isi_roi_index_t
```

ISI roi index number.

```
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 adddr
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.

2.30 Common Driver

FSL_COMMON_DRIVER_VERSION
common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE
No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART
Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART
Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM
Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART
Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true**, else return **false**.

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

SDK_SIZEALIGN(var, alignbytes)

Macro to define a variable with L1 d-cache line size alignment

Macro to define a variable with L2 cache line size alignment

Macro to change a value to a given size aligned value

AT_NONCACHEABLE_SECTION(var)

Define a variable *var*, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_NONCACHEABLE_SECTION_INIT(var)

Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_CACHE_LINE_SECTION(var)

Define a variable *var*, which is cache line size aligned and be placed in CacheLineData section.

AT_CACHE_LINE_SECTION_INIT(var)

Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

AT_QUICKACCESS_SECTION_CODE(func)

Place function in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA(var)

Place data in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA_ALIGN(var, alignbytes)

Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

RAMFUNCTION_SECTION_CODE(func)

Place function in ram.

enum _status_groups

Status group numbers.

Values:

enumerator kStatusGroup_Generic

Group number for generic status codes.

enumerator kStatusGroup_FLASH

Group number for FLASH status codes.

enumerator kStatusGroup_LPSPI

Group number for LPSPI status codes.

enumerator kStatusGroup_FLEXIO_SPI

Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI

Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
Group number for LPI2C status codes.

enumerator kStatusGroup_UART
Group number for UART status codes.

enumerator kStatusGroup_I2C
Group number for UART status codes.

enumerator kStatusGroup_LPSCI
Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
Group number for LPUART status codes.

enumerator kStatusGroup_SPI
Group number for SPI status code.

enumerator kStatusGroup_XRDC
Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPIDSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
Group number for debug console status codes.

enumerator kStatusGroup_SEMC
Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
Starting number for application groups.

enumerator kStatusGroup_IAP
Group number for IAP status codes

enumerator kStatusGroup_SFA
Group number for SFA status codes

enumerator kStatusGroup_SPC
Group number for SPC status codes.

enumerator kStatusGroup_PUF
Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes

enumerator kStatusGroup_VBAT
Group number for VBAT status codes

enumerator kStatusGroup_XSPI
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator `kStatusGroup_HAL_RNG`
Group number for HAL RNG status codes.

enumerator `kStatusGroup_HAL_I2S`
Group number for HAL I2S status codes.

enumerator `kStatusGroup_HAL_ADC_SENSOR`
Group number for HAL ADC SENSOR status codes.

enumerator `kStatusGroup_TIMERMANAGER`
Group number for TiMER MANAGER status codes.

enumerator `kStatusGroup_SERIALMANAGER`
Group number for SERIAL MANAGER status codes.

enumerator `kStatusGroup_LED`
Group number for LED status codes.

enumerator `kStatusGroup_BUTTON`
Group number for BUTTON status codes.

enumerator `kStatusGroup_EXTERN_EEPROM`
Group number for EXTERN EEPROM status codes.

enumerator `kStatusGroup_SHELL`
Group number for SHELL status codes.

enumerator `kStatusGroup_MEM_MANAGER`
Group number for MEM MANAGER status codes.

enumerator `kStatusGroup_LIST`
Group number for List status codes.

enumerator `kStatusGroup_OSA`
Group number for OSA status codes.

enumerator `kStatusGroup_COMMON_TASK`
Group number for Common task status codes.

enumerator `kStatusGroup_MSG`
Group number for messaging status codes.

enumerator `kStatusGroup_SDK_OCOTP`
Group number for OCOTP status codes.

enumerator `kStatusGroup_SDK_FLEXSPINOR`
Group number for FLEXSPINOR status codes.

enumerator `kStatusGroup_CODEC`
Group number for codec status codes.

enumerator `kStatusGroup_ASRC`
Group number for codec status ASRC.

enumerator `kStatusGroup_OTFAD`
Group number for codec status codes.

enumerator `kStatusGroup_SDIOSLV`
Group number for SDIOSLV status codes.

enumerator `kStatusGroup_MECC`
Group number for MECC status codes.

enumerator kStatusGroup_ENET_QOS
Group number for ENET_QOS status codes.

enumerator kStatusGroup_LOG
Group number for LOG status codes.

enumerator kStatusGroup_I3CBUS
Group number for I3CBUS status codes.

enumerator kStatusGroup_QSCI
Group number for QSCI status codes.

enumerator kStatusGroup_ELEMU
Group number for ELEMU status codes.

enumerator kStatusGroup_QUEUEDSPI
Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC

Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT

Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT

Group number for A-format status codes.

Generic status return codes.

Values:

enumerator kStatus_Success

Generic status for Success.

enumerator kStatus_Fail

Generic status for Fail.

enumerator kStatus_ReadOnly

Generic status for read only failure.

enumerator kStatus_OutOfRange

Generic status for out of range access.

enumerator kStatus_InvalidArgument

Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

void SDK_Free(void *ptr)

Free memory.

Parameters

- ptr – The memory to be release.

```
void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)
```

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- delayTime_us – Delay time in unit of microsecond.
- coreClock_Hz – Core clock frequency with Hz.

```
static inline status_t EnableIRQ(IRQn_Type interrupt)
```

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt enabled successfully
- kStatus_Fail – Failed to enable the interrupt

```
static inline status_t DisableIRQ(IRQn_Type interrupt)
```

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt disabled successfully
- kStatus_Fail – Failed to disable the interrupt

```
static inline status_t EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)
```

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- `kStatus_Success` – Interrupt priority set successfully
- `kStatus_Fail` – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ to set.
- `priNum` – Priority number set to interrupt controller register.

Return values

- `kStatus_Success` – Interrupt priority set successfully
- `kStatus_Fail` – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The flag which IRQ to clear.

Return values

- `kStatus_Success` – Interrupt priority set successfully
- `kStatus_Fail` – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the `EnableGlobalIRQ()`.

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the `EnableGlobalIRQ()` and `DisableGlobalIRQ()` in pair.

Parameters

- `primask` – value of primask register to be restored. The primask value is supposed to be provided by the `DisableGlobalIRQ()`.

```
static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t
                                                    newValue)
```

```
static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)
```

`FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ`

Macro to use the default weak IRQ handler in drivers.

`MAKE_STATUS(group, code)`

Construct a status code value from a group and code number.

`MAKE_VERSION(major, minor, bugfix)`

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major	Version	Minor	Version	Bug Fix
31	25	24	17	16	9 8 0

`ARRAY_SIZE(x)`

Computes the number of elements in an array.

`UINT64_H(X)`

Macro to get upper 32 bits of a 64-bit value

`UINT64_L(X)`

Macro to get lower 32 bits of a 64-bit value

`SUPPRESS_FALL_THROUGH_WARNING()`

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag `-Wextra` or `-Wimplicit-fallthrough=n` when using `armgcc`. To suppress this warning, “`SUPPRESS_FALL_THROUGH_WARNING()`,” need to be added at the end of each case section which misses “break;”statement.

`MSDK_REG_SECURE_ADDR(x)`

Convert the register address to the one used in secure mode.

`MSDK_REG_NONSECURE_ADDR(x)`

Convert the register address to the one used in non-secure mode.

`MSDK_HAS_DWT_CYCCNT`

The chip supports DWT CYCCNT or not.

`MSDK_INVALID_IRQ_HANDLER`

Invalid IRQ handler address.

2.31 LCDIFv3: LCD Interface v3

```
void LCDIFV3_Init(LCDIF_Type *base)
```

Initializes the LCDIF v3.

This function ungates the LCDIF v3 clock and release the peripheral reset.

Parameters

- `base` – LCDIF v3 peripheral base address.

void LCDIFV3_Deinit(LCDIF_Type *base)

Deinitializes the LCDIF peripheral.

Parameters

- base – LCDIF peripheral base address.

void LCDIFV3_Reset(LCDIF_Type *base)

Reset the LCDIF v3.

Parameters

- base – LCDIF peripheral base address.

void LCDIFV3_DisplayGetDefaultConfig(*lcdifv3_display_config_t* *config)

Gets the LCDIF display default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values.

```
config->panelWidth    = 0U;
config->panelHeight   = 0U;
config->hsw            = 3U;
config->hfp            = 3U;
config->hbp            = 3U;
config->vsw            = 3U;
config->vfp            = 3U;
config->vbp            = 3U;
config->polarityFlags = kLCDIFV3_VsyncActiveHigh | kLCDIFV3_HsyncActiveHigh | kLCDIFV3_
↪DataEnableActiveHigh |
                      kLCDIFV3_DriveDataOnRisingClkEdge | kLCDIFV3_DataActiveHigh;
config->lineOrder      = kLCDIFV3_LineOrderRGB;
```

Parameters

- config – Pointer to the LCDIF configuration structure.

void LCDIFV3_SetDisplayConfig(LCDIF_Type *base, const *lcdifv3_display_config_t* *config)

Set the LCDIF v3 display configurations.

Parameters

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

static inline void LCDIFV3_EnableDisplay(LCDIF_Type *base, bool enable)

Enable or disable the display.

Parameters

- base – LCDIF peripheral base address.
- enable – Enable or disable.

static inline void LCDIFV3_EnableInterrupts(LCDIF_Type *base, uint32_t mask)

Enables LCDIF interrupt requests.

Parameters

- base – LCDIF peripheral base address.
- mask – interrupt source, OR'ed value of `_lcdifv3_interrupt`.

static inline void LCDIFV3_DisableInterrupts(LCDIF_Type *base, uint8_t domain, uint32_t mask)

Disables LCDIF interrupt requests.

Parameters

- base – LCDIF peripheral base address.
- domain – CPU domain the interrupt signal routed to.
- mask – interrupt source, OR'ed value of `_lcdifv3_interrupt`.

static inline uint32_t LCDIFV3_GetInterruptStatus(LCDIF_Type *base)

Get LCDIF interrupt pending status.

Parameters

- base – LCDIF peripheral base address.

Returns

Interrupt pending status, OR'ed value of `_lcdifv3_interrupt`.

static inline void LCDIFV3_ClearInterruptStatus(LCDIF_Type *base, uint32_t mask)

Clear LCDIF interrupt pending status.

Parameters

- base – LCDIF peripheral base address.
- mask – of the flags to clear, OR'ed value of `_lcdifv3_interrupt`.

static inline void LCDIFV3_SetLayerSize(LCDIF_Type *base, uint8_t layerIndex, uint16_t width, uint16_t height)

Set the layer dimension.

Note: The layer width must be in multiples of the number of pixels that can be stored in 32 bits

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Layer layerIndex.
- width – Layer width in pixel.
- height – Layer height.

void LCDIFV3_SetLayerBufferConfig(LCDIF_Type *base, uint8_t layerIndex, const *lcdifv3_buffer_config_t* *config)

Set the layer source buffer configuration.

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Layer layerIndex.
- config – Pointer to the configuration.

static inline void LCDIFV3_SetLayerBufferAddr(LCDIF_Type *base, uint8_t layerIndex, uint32_t addr)

Set the layer source buffer address.

This function is used for fast runtime source buffer change.

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Layer layerIndex.
- addr – The new source buffer address passed to the layer, should be 64-bit aligned.

```
static inline void LCDIFV3_EnableLayer(LCDIF_Type *base, uint8_t layerIndex, bool enable)
```

Enable or disable the layer.

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Layer layerIndex.
- enable – Pass in true to enable, false to disable.

```
static inline void LCDIFV3_TriggerLayerShadowLoad(LCDIF_Type *base, uint8_t layerIndex)
```

Trigger the layer configuration shadow load.

The new layer configurations are written to the shadow registers first, When all configurations written finished, call this function, then shadowed control registers are updated to the active control registers on VSYNC of next frame.

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Layer layerIndex.

```
void LCDIFV3_SetCscMode(LCDIF_Type *base, lcdifv3_csc_mode_t mode)
```

Set the color space conversion mode.

Supports YUV2RGB and YCbCr2RGB.

Parameters

- base – LCDIFv3 peripheral base address.
- layerIndex – Index of the layer.
- mode – The conversion mode.

```
void LCDIFV3_EnablePlanePanic(LCDIF_Type *base)
```

Enable plane panic to avoid underrun.

Parameters

- base – LCDIFv3 peripheral base address.

```
FSL_LCDIFV3_DRIVER_VERSION
```

LCDIF v3 driver version.

```
enum _lcdifv3_polarity_flags
```

LCDIF v3 signal polarity flags.

Values:

```
enumerator kLCDIFV3_VsyncActiveHigh
```

VSYNC active high.

```
enumerator kLCDIFV3_HsyncActiveHigh
```

HSYNC active high.

```
enumerator kLCDIFV3_DataEnableActiveHigh
```

Data enable line active high.

```
enumerator kLCDIFV3_DriveDataOnRisingClkEdge
```

Output data on rising clock edge, capture data on falling clock edge.

```
enumerator kLCDIFV3_DataActiveHigh
```

Data active high.

enumerator kLCDIFV3__VsyncActiveLow
VSYNC active low.

enumerator kLCDIFV3__HsyncActiveLow
HSYNC active low.

enumerator kLCDIFV3__DataEnableActiveLow
Data enable line active low.

enumerator kLCDIFV3__DriveDataOnFallingClkEdge
Output data on falling clock edge, capture data on rising clock edge.

enumerator kLCDIFV3__DataActiveLow
Data active high.

enum _lcdifv3__interrupt
The LCDIF v3 interrupts.

Values:

enumerator kLCDIFV3__Layer0FifoEmptyInterrupt
Layer 0 FIFO empty.

enumerator kLCDIFV3__Layer1FifoEmptyInterrupt
Layer 1 FIFO empty.

enumerator kLCDIFV3__Layer2FifoEmptyInterrupt
Layer 2 FIFO empty.

enumerator kLCDIFV3__Layer3FifoEmptyInterrupt
Layer 3 FIFO empty.

enumerator kLCDIFV3__Layer4FifoEmptyInterrupt
Layer 4 FIFO empty.

enumerator kLCDIFV3__Layer5FifoEmptyInterrupt
Layer 5 FIFO empty.

enumerator kLCDIFV3__Layer6FifoEmptyInterrupt
Layer 6 FIFO empty.

enumerator kLCDIFV3__Layer7FifoEmptyInterrupt
Layer 7 FIFO empty.

enumerator kLCDIFV3__Layer0DmaDoneInterrupt
Layer 0 DMA done.

enumerator kLCDIFV3__Layer1DmaDoneInterrupt
Layer 1 DMA done.

enumerator kLCDIFV3__Layer2DmaDoneInterrupt
Layer 2 DMA done.

enumerator kLCDIFV3__Layer3DmaDoneInterrupt
Layer 3 DMA done.

enumerator kLCDIFV3__Layer4DmaDoneInterrupt
Layer 4 DMA done.

enumerator kLCDIFV3__Layer5DmaDoneInterrupt
Layer 5 DMA done.

enumerator kLCDIFV3__Layer6DmaDoneInterrupt
Layer 6 DMA done.

enumerator kLCDIFV3__Layer7DmaDoneInterrupt
Layer 7 DMA done.

enumerator kLCDIFV3__Layer0DmaErrorInterrupt
Layer 0 DMA error.

enumerator kLCDIFV3__Layer1DmaErrorInterrupt
Layer 1 DMA error.

enumerator kLCDIFV3__Layer2DmaErrorInterrupt
Layer 2 DMA error.

enumerator kLCDIFV3__Layer3DmaErrorInterrupt
Layer 3 DMA error.

enumerator kLCDIFV3__Layer4DmaErrorInterrupt
Layer 4 DMA error.

enumerator kLCDIFV3__Layer5DmaErrorInterrupt
Layer 5 DMA error.

enumerator kLCDIFV3__Layer6DmaErrorInterrupt
Layer 6 DMA error.

enumerator kLCDIFV3__Layer7DmaErrorInterrupt
Layer 7 DMA error.

enumerator kLCDIFV3__VerticalBlankingInterrupt
Start of vertical blanking period.

enumerator kLCDIFV3__OutputUnderrunInterrupt
Output buffer underrun.

enumerator kLCDIFV3__VsyncEdgeInterrupt
Interrupt at VSYNC edge.

enum _lcdifv3_line_order
The LCDIF v3 output line order.

Values:

enumerator kLCDIFV3__LineOrderRGBOrYUV
RGB/YUV

enumerator kLCDIFV3__LineOrderRBG
RBG

enumerator kLCDIFV3__LineOrderGBR
GBR

enumerator kLCDIFV3__LineOrderGRBOrUYV
GRB/UYV

enumerator kLCDIFV3__LineOrderBRG
BRG

enumerator kLCDIFV3__LineOrderBGR
BGR

enumerator kLCDIFV3__LineOrderRGB555
RGB555

enumerator kLCDIFV3__LineOrderRGB565
RGB565

enumerator kLCDIFV3__LineOrderYUYVAt15To0
YUYV at [15:0]

enumerator kLCDIFV3__LineOrderUYVYAt15To0
UYVY at [15:0]

enumerator kLCDIFV3__LineOrderYVYUAt15To0
YVYU at [15:0]

enumerator kLCDIFV3__LineOrderVYUYAt15To0
VYUY at [15:0]

enumerator kLCDIFV3__LineOrderYUYVAt23To8
YUYV at [23:8]

enumerator kLCDIFV3__LineOrderUYVYAt23To8
UYVY at [23:8]

enumerator kLCDIFV3__LineOrderYVYUAt23To8
YVYU at [23:8]

enumerator kLCDIFV3__LineOrderVYUYAt23To8
VYUY at [23:8]

enum _lcdifv3__csc__mode

LCDIF v3 color space conversion mode.

Values:

enumerator kLCDIFV3__CscYUV2RGB
YUV to RGB.

enumerator kLCDIFV3__CscYCbCr2RGB
YCbCr to RGB.

enumerator kLCDIFV3__CscRGB2YUV
RGB to YUV.

enumerator kLCDIFV3__CscRGB2YCbCr
RGB to YCbCr.

enum _lcdifv3__pixel__format

LCDIF v3 pixel format.

Values:

enumerator kLCDIFV3__PixelFormatRGB565
RGB565, two pixels use 32 bits.

enumerator kLCDIFV3__PixelFormatARGB1555
ARGB1555, two pixels use 32 bits.

enumerator kLCDIFV3__PixelFormatARGB4444
ARGB4444, two pixels use 32 bits.

enumerator kLCDIFV3__PixelFormatYVYU
YVYU {Y2,V1,Y1,U1}

```
enumerator kLCDIFV3_PixelFormatYUYV
    YUYV {Y2,U1,Y1,V1}
enumerator kLCDIFV3_PixelFormatVYUY
    VYUY {V1,Y2,U1,Y1}
enumerator kLCDIFV3_PixelFormatUYVY
    UYVY {U1,Y2,V1,Y1}
enumerator kLCDIFV3_PixelFormatRGB888
    RGB888 packed, one pixel uses 24 bits.
enumerator kLCDIFV3_PixelFormatARGB8888
    ARGB8888 unpacked, one pixel uses 32 bits.
enumerator kLCDIFV3_PixelFormatABGR8888
    ABGR8888 unpacked, one pixel uses 32 bits.
enum _lcdifv3_pd_alpha_mode
    LCDIF v3 PoterDuff alpha mode.
    Values:
    enumerator kLCDIFV3_PD_AlphaStraight
        Straight mode.
    enumerator kLCDIFV3_PD_AlphaInversed
        Inversed mode.
typedef enum _lcdifv3_line_order lcdifv3_line_order_t
    The LCDIF v3 output line order.
typedef struct _lcdifv3_display_config lcdifv3_display_config_t
    LCDIF v3 display configure structure.
typedef enum _lcdifv3_csc_mode lcdifv3_csc_mode_t
    LCDIF v3 color space conversion mode.
typedef enum _lcdifv3_pixel_format lcdifv3_pixel_format_t
    LCDIF v3 pixel format.
typedef struct _lcdifv3_buffer_config lcdifv3_buffer_config_t
    LCDIF v3 source buffer configuration.
typedef enum _lcdifv3_pd_alpha_mode lcdifv3_pd_alpha_mode_t
    LCDIF v3 PoterDuff alpha mode.
LCDIFV3_ADDR_CPU_2_IP(addr)
LCDIFV3_MAKE_FIFO_EMPTY_INTERRUPT(layer)
    LCDIF v3 FIFO empty interrupt.
LCDIFV3_MAKE_DMA_DONE_INTERRUPT(layer)
    LCDIF v3 DMA done interrupt.
LCDIFV3_MAKE_DMA_ERROR_INTERRUPT(layer)
    LCDIF v3 DMA error interrupt.
struct _lcdifv3_display_config
    #include <fsl_lcdifv3.h> LCDIF v3 display configure structure.
```


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 `_lcdifv3_polarity_flags`, used to contol the signal polarity.

lcdifv3_line_order_t lineOrder

Line order.

struct `_lcdifv3_buffer_config`

#include <fsl_lcdifv3.h> LCDIF v3 source buffer configuration.

Public Members

lcdifv3_pixel_format_t pixelFormat

Source buffer pixel format.

2.32 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_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

enum _lin_lpuart_interrupt_enable

Values:

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.

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_RxRingBufferOverflow
    LPUART RX software ring buffer overrun.
enumerator kStatus_LPUART_RxHardwareOverflow
    LPUART RX receiver overrun.
enumerator kStatus_LPUART_NoiseError
    LPUART noise error.
enumerator kStatus_LPUART_FramingError
    LPUART framing error.
enumerator kStatus_LPUART_ParityError
    LPUART parity error.
enum lin_lpuart_bit_count_per_char_t
    Values:
    enumerator LPUART_8_BITS_PER_CHAR
        8-bit data characters
    enumerator LPUART_9_BITS_PER_CHAR
        9-bit data characters
    enumerator LPUART_10_BITS_PER_CHAR
        10-bit data characters
typedef enum lin_lpuart_stop_bit_count lin_lpuart_stop_bit_count_t
static inline bool LIN_LPUART_GetRxDataPolarity(const LPUART_Type *base)
static inline void LIN_LPUART_SetRxDataPolarity(LPUART_Type *base, bool polarity)
static inline void LIN_LPUART_WriteByte(LPUART_Type *base, uint8_t data)
static inline void LIN_LPUART_ReadByte(const LPUART_Type *base, uint8_t *readData)
```

```
status_t LIN_LPUART_CalculateBaudRate(LPUART_Type *base, uint32_t baudRate_Bps,  
                                     uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)
```

Calculates the best osr and sbr value for configured baudrate.

Parameters

- base – LPUART peripheral base address
- baudRate_Bps – user configuration structure of type #lin_user_config_t
- srcClock_Hz – pointer to the LIN_LPUART driver state structure
- osr – pointer to osr value
- sbr – pointer to sbr value

Returns

An error code or lin_status_t

```
void LIN_LPUART_SetBaudRate(LPUART_Type *base, uint32_t *osr, uint16_t *sbr)
```

Configure baudrate according to osr and sbr value.

Parameters

- base – LPUART peripheral base address
- osr – pointer to osr value
- sbr – pointer to sbr value

```
lin_status_t LIN_LPUART_Init(LPUART_Type *base, lin_user_config_t *linUserConfig,  
                             lin_state_t *linCurrentState, uint32_t linSourceClockFreq)
```

Initializes an LIN_LPUART instance for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN_LPUART clock source in the application to initialize the LIN_LPUART. This function initializes a LPUART instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, set break field length to be 13 bit times minimum, enable the break detect interrupt, Rx complete interrupt, frame error detect interrupt, and enable the LPUART module transmitter and receiver

Parameters

- base – LPUART peripheral base address
- linUserConfig – user configuration structure of type #lin_user_config_t
- linCurrentState – pointer to the LIN_LPUART driver state structure

Returns

An error code or lin_status_t

```
lin_status_t LIN_LPUART_Deinit(LPUART_Type *base)
```

Shuts down the LIN_LPUART by disabling interrupts and transmitter/receiver.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

```
lin_status_t LIN_LPUART_SendFrameDataBlocking(LPUART_Type *base, const uint8_t *txBuff,  
                                              uint8_t txSize, uint32_t timeoutMSec)
```

Sends Frame data out through the LIN_LPUART module using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendFrameData(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN_LPUART module using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_GetTransmitStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. This function return LIN_TX_BUSY while sending, or LIN_TIMEOUT if timeout has occurred, or return LIN_SUCCESS when the transmission is complete. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to transmit

Returns

lin_status_t LIN_TX_BUSY, LIN_SUCCESS or LIN_TIMEOUT

lin_status_t LIN_LPUART_RecvFrmDataBlocking(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN_LPUART module using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

`lin_status_t LIN_LPUART_RecvFrmData(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize)`

Receives frame data through the LIN_LPUART module using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Parameters

- `base` – LPUART peripheral base address
- `rxBuff` – buffer containing 8-bit received data
- `rxSize` – the number of bytes to receive

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_AbortTransferData(LPUART_Type *base)`

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GetReceiveStatus(LPUART_Type *base, uint8_t *bytesRemaining)`

Get status of an on-going non-blocking reception. While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function returns the current event ID, `LIN_RX_BUSY` while receiving and return `LIN_SUCCESS`, or timeout (`LIN_TIMEOUT`) when the reception is complete. The `bytesRemaining` shows number of bytes that still needed to receive.

Parameters

- `base` – LPUART peripheral base address
- `bytesRemaining` – Number of bytes still needed to receive

Returns

`lin_status_t` `LIN_RX_BUSY`, `LIN_TIMEOUT` or `LIN_SUCCESS`

`lin_status_t LIN_LPUART_GoToSleepMode(LPUART_Type *base)`

This function puts current node to sleep mode. This function changes current node state to `LIN_NODE_STATE_SLEEP_MODE`.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GotoIdleState(LPUART_Type *base)`

Puts current LIN node to Idle state. This function changes current node state to `LIN_NODE_STATE_IDLE`.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

lin_status_t LIN_LPUART_SendWakeupSignal(LPUART_Type *base)

Sends a wakeup signal through the LIN_LPUART interface.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_MasterSendHeader(LPUART_Type *base, uint8_t id)

Sends frame header out through the LIN_LPUART module using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- base – LPUART peripheral base address
- id – Frame Identifier

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_EnableIRQ(LPUART_Type *base)

Enables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_DisableIRQ(LPUART_Type *base)

Disables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns

lin_status_t

void LIN_LPUART_IRQHandler(LPUART_Type *base)

LIN_LPUART RX TX interrupt handler.

Parameters

- base – LPUART peripheral base address

Returns

void

LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT

Max loops to wait for LPUART transmission complete.

When de-initializing the LIN LPUART module, the program shall wait for the previous transmission to complete. This parameter defines how many loops to check completion before return error. If defined as 0, driver will wait forever until completion.

AUTOBAUD_BAUDRATE_TOLERANCE

BIT_RATE_TOLERANCE_UNSYNC

BIT_DURATION_MAX_19200

BIT_DURATION_MIN_19200

BIT_DURATION_MAX_14400

BIT_DURATION_MIN_14400

BIT_DURATION_MAX_9600

BIT_DURATION_MIN_9600

BIT_DURATION_MAX_4800

BIT_DURATION_MIN_4800

BIT_DURATION_MAX_2400

BIT_DURATION_MIN_2400

TWO_BIT_DURATION_MAX_19200

TWO_BIT_DURATION_MIN_19200

TWO_BIT_DURATION_MAX_14400

TWO_BIT_DURATION_MIN_14400

TWO_BIT_DURATION_MAX_9600

TWO_BIT_DURATION_MIN_9600

TWO_BIT_DURATION_MAX_4800

TWO_BIT_DURATION_MIN_4800

TWO_BIT_DURATION_MAX_2400

TWO_BIT_DURATION_MIN_2400

AUTOBAUD_BREAK_TIME_MIN

2.33 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.

2.34 LPI2C Master Driver

`void LPI2C_MasterGetDefaultConfig(lpi2c_master_config_t *masterConfig)`

Provides a default configuration for the LPI2C master peripheral.

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->debugEnable       = false;
masterConfig->ignoreAck         = false;
masterConfig->pinConfig          = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz       = 100000U;
masterConfig->busIdleTimeout_ns = 0;
masterConfig->pinLowTimeout_ns  = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable = false;
masterConfig->hostRequest.source  = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `LPI2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `lpi2c_master_config_t`.

`void LPI2C_MasterInit(LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t sourceClock_Hz)`

Initializes the LPI2C master peripheral.

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `LPI2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`void LPI2C_MasterDeinit(LPI2C_Type *base)`

Deinitializes the LPI2C master peripheral.

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

`void LPI2C_MasterConfigureDataMatch(LPI2C_Type *base, const lpi2c_data_match_config_t *matchConfig)`

Configures LPI2C master data match feature.

Parameters

- `base` – The LPI2C peripheral base address.
- `matchConfig` – Settings for the data match feature.

status_t LPI2C_MasterCheckAndClearError(LPI2C_Type *base, uint32_t status)

Convert provided flags to status code, and clear any errors if present.

Parameters

- base – The LPI2C peripheral base address.
- status – Current status flags value that will be checked.

Return values

- kStatus_Success –
- kStatus_LPI2C_PinLowTimeout –
- kStatus_LPI2C_ArbitrationLost –
- kStatus_LPI2C_Nak –
- kStatus_LPI2C_FifoError –

status_t LPI2C_CheckForBusyBus(LPI2C_Type *base)

Make sure the bus isn't already busy.

A busy bus is allowed if we are the one driving it.

Parameters

- base – The LPI2C peripheral base address.

Return values

- kStatus_Success –
- kStatus_LPI2C_Busy –

static inline void LPI2C_MasterReset(LPI2C_Type *base)

Performs a software reset.

Restores the LPI2C master peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_MasterEnable(LPI2C_Type *base, bool enable)

Enables or disables the LPI2C module as master.

Parameters

- base – The LPI2C peripheral base address.
- enable – Pass true to enable or false to disable the specified LPI2C as master.

static inline uint32_t LPI2C_MasterGetStatusFlags(LPI2C_Type *base)

Gets the LPI2C master status flags.

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_lpi2c_master_flags`

Parameters

- base – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void LPI2C_MasterClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)

Clears the LPI2C master status flag state.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

See also:

`_lpi2c_master_flags`.

Parameters

- base – The LPI2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_MasterGetStatusFlags()`.

static inline void LPI2C_MasterEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Enables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline void LPI2C_MasterDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Disables the LPI2C master interrupt requests.

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_lpi2c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t LPI2C_MasterGetEnabledInterrupts(LPI2C_Type *base)

Returns the set of currently enabled LPI2C master interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline void LPI2C_MasterEnableDMA(LPI2C_Type *base, bool enableTx, bool enableRx)
```

Enables or disables LPI2C master DMA requests.

Parameters

- `base` – The LPI2C peripheral base address.
- `enableTx` – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- `enableRx` – Enable flag for receive DMA request. Pass true for enable, false for disable.

```
static inline uint32_t LPI2C_MasterGetTxFifoAddress(LPI2C_Type *base)
```

Gets LPI2C master transmit data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

```
static inline uint32_t LPI2C_MasterGetRxFifoAddress(LPI2C_Type *base)
```

Gets LPI2C master receive data register address for DMA transfer.

Parameters

- `base` – The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

```
static inline void LPI2C_MasterSetWatermarks(LPI2C_Type *base, size_t txWords, size_t rxWords)
```

Sets the watermarks for LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txWords` – Transmit FIFO watermark value in words. The `kLPI2C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO is equal or less than `txWords`. Writing a value equal or greater than the FIFO size is truncated.
- `rxWords` – Receive FIFO watermark value in words. The `kLPI2C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO is greater than `rxWords`. Writing a value equal or greater than the FIFO size is truncated.

```
static inline void LPI2C_MasterGetFifoCounts(LPI2C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of words in the LPI2C master FIFOs.

Parameters

- `base` – The LPI2C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.

```
void LPI2C_MasterSetBaudRate(LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t
                             baudRate_Hz)
```

Sets the I2C bus frequency for master transactions.

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Note: Please note that the second parameter is the clock frequency of LPI2C module, the third parameter means user configured bus baudrate, this implementation is different from other I2C drivers which use baudrate configuration as second parameter and source clock frequency as third parameter.

Parameters

- `base` – The LPI2C peripheral base address.
- `sourceClock_Hz` – LPI2C functional clock frequency in Hertz.
- `baudRate_Hz` – Requested bus frequency in Hertz.

```
static inline bool LPI2C_MasterGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

```
status_t LPI2C_MasterStart(LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
```

Sends a START signal and slave address on the I2C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

```
static inline status_t LPI2C_MasterRepeatedStart(LPI2C_Type *base, uint8_t address,
                                                  lpi2c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `LPI2C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The LPI2C peripheral base address.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kLPI2C_Read` or `kLPI2C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.

status_t LPI2C_MasterSend(LPI2C_Type *base, void *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_LPI2C_Nak`.

Parameters

- `base` – The LPI2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or over run.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The LPI2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.

- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterStop(LPI2C_Type *base)`

Sends a STOP signal on the I2C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The LPI2C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`status_t LPI2C_MasterTransferBlocking(LPI2C_Type *base, lpi2c_master_transfer_t *transfer)`

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- `base` – The LPI2C peripheral base address.
- `transfer` – Pointer to the transfer structure.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_LPI2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_LPI2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_LPI2C_FifoError` – FIFO under run or overrun.
- `kStatus_LPI2C_ArbitrationLost` – Arbitration lost error.
- `kStatus_LPI2C_PinLowTimeout` – SCL or SDA were held low longer than the timeout.

`void LPI2C_MasterTransferCreateHandle(LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)`

Creates a new handle for the LPI2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `LPI2C_MasterTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_MasterTransferNonBlocking(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *lpi2c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_LPI2C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t LPI2C_MasterTransferGetCount(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *size_t* *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void LPI2C_MasterTransferAbort(LPI2C_Type *base, *lpi2c_master_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

void LPI2C_MasterTransferHandleIRQ(LPI2C_Type *base, void *lpi2cMasterHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The LPI2C peripheral base address.
- lpi2cMasterHandle – Pointer to the LPI2C master driver handle.

enum _lpi2c_master_flags

LPI2C master peripheral flags.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kLPI2C_MasterTxReadyFlag

Transmit data flag

enumerator kLPI2C_MasterRxReadyFlag

Receive data flag

enumerator kLPI2C_MasterEndOfPacketFlag

End Packet flag

enumerator kLPI2C_MasterStopDetectFlag

Stop detect flag

enumerator kLPI2C_MasterNackDetectFlag

NACK detect flag

enumerator kLPI2C_MasterArbitrationLostFlag

Arbitration lost flag

enumerator kLPI2C_MasterFifoErrFlag

FIFO error flag

enumerator kLPI2C_MasterPinLowTimeoutFlag

Pin low timeout flag

enumerator kLPI2C_MasterDataMatchFlag

Data match flag

enumerator kLPI2C_MasterBusyFlag

Master busy flag

enumerator kLPI2C_MasterBusBusyFlag

Bus busy flag

enumerator kLPI2C_MasterClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_MasterIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_MasterErrorFlags

Errors to check for.

enum _lpi2c_direction

Direction of master and slave transfers.

Values:

enumerator kLPI2C_Write

Master transmit.

enumerator kLPI2C_Read

Master receive.

enum _lpi2c_master_pin_config

LPI2C pin configuration.

Values:

enumerator kLPI2C_2PinOpenDrain

LPI2C Configured for 2-pin open drain mode

enumerator kLPI2C_2PinOutputOnly

LPI2C Configured for 2-pin output only mode (ultra-fast mode)

enumerator kLPI2C_2PinPushPull

LPI2C Configured for 2-pin push-pull mode

enumerator kLPI2C_4PinPushPull

LPI2C Configured for 4-pin push-pull mode

enumerator kLPI2C_2PinOpenDrainWithSeparateSlave

LPI2C Configured for 2-pin open drain mode with separate LPI2C slave

enumerator kLPI2C_2PinOutputOnlyWithSeparateSlave

LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave

enumerator kLPI2C_2PinPushPullWithSeparateSlave

LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave

enumerator kLPI2C_4PinPushPullWithInvertedOutput

LPI2C Configured for 4-pin push-pull mode(inverted outputs)

enum _lpi2c_host_request_source

LPI2C master host request selection.

Values:

enumerator kLPI2C_HostRequestExternalPin

Select the LPI2C_HREQ pin as the host request input

enumerator kLPI2C_HostRequestInputTrigger

Select the input trigger as the host request input

enum _lpi2c_host_request_polarity

LPI2C master host request pin polarity configuration.

Values:

enumerator kLPI2C_HostRequestPinActiveLow

Configure the LPI2C_HREQ pin active low

enumerator kLPI2C_HostRequestPinActiveHigh

Configure the LPI2C_HREQ pin active high

enum _lpi2c_data_match_config_mode

LPI2C master data match configuration modes.

Values:

enumerator kLPI2C_MatchDisabled

LPI2C Match Disabled

enumerator kLPI2C_1stWordEqualsM0OrM1

LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1

enumerator kLPI2C_AnyWordEqualsM0OrM1

LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1

enumerator kLPI2C_1stWordEqualsM0And2ndWordEqualsM1

LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1

enumerator kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1

LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1

enumerator kLPI2C_1stWordAndM1EqualsM0AndM1

LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1

enumerator kLPI2C_AnyWordAndM1EqualsM0AndM1

LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1

enum _lpi2c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_lpi2c_master_transfer::flags` field.

Values:

enumerator kLPI2C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kLPI2C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kLPI2C_TransferRepeatedStartFlag

Send a repeated start condition

enumerator kLPI2C_TransferNoStopFlag

Don't send a stop condition.

```
typedef enum _lpi2c_direction lpi2c_direction_t
```

Direction of master and slave transfers.

```
typedef enum _lpi2c_master_pin_config lpi2c_master_pin_config_t
```

LPI2C pin configuration.

```
typedef enum _lpi2c_host_request_source lpi2c_host_request_source_t
```

LPI2C master host request selection.

```
typedef enum _lpi2c_host_request_polarity lpi2c_host_request_polarity_t
```

LPI2C master host request pin polarity configuration.

```
typedef struct _lpi2c_master_config lpi2c_master_config_t
```

Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the `LPI2C_MasterGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _lpi2c_data_match_config_mode lpi2c_data_match_config_mode_t
```

LPI2C master data match configuration modes.

```
typedef struct _lpi2c_match_config lpi2c_data_match_config_t
```

LPI2C master data match configuration structure.

```
typedef struct _lpi2c_master_transfer lpi2c_master_transfer_t
```

LPI2C master descriptor of the transfer.

```
typedef struct _lpi2c_master_handle lpi2c_master_handle_t
```

LPI2C master handle of the transfer.

```
typedef void (*lpi2c_master_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to `LPI2C_MasterTransferCreateHandle()`.

Param base

The LPI2C peripheral base address.

Param handle

Pointer to the LPI2C master driver handle.

Param completionStatus

Either `kStatus_Success` or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef void (*lpi2c_master_isr_t)(LPI2C_Type *base, void *handle)
```

Typedef for master interrupt handler, used internally for LPI2C master interrupt and EDMA transactional APIs.

```
struct _lpi2c_master_config
```

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the `LPI2C_MasterGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

bool enableDoze

Whether master is enabled in doze mode.

bool debugEnable

Enable transfers to continue when halted in debug mode.

bool ignoreAck

Whether to ignore ACK/NACK.

lpi2c_master_pin_config_t pinConfig

The pin configuration option.

uint32_t baudRate_Hz

Desired baud rate in Hertz.

uint32_t busIdleTimeout_ns

Bus idle timeout in nanoseconds. Set to 0 to disable.

uint32_t pinLowTimeout_ns

Pin low timeout in nanoseconds. Set to 0 to disable.

uint8_t sdaGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SDA pin. Set to 0 to disable.

uint8_t sclGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SCL pin. Set to 0 to disable.

struct *_lpi2c_master_config* hostRequest

Host request options.

struct *_lpi2c_match_config*

#include <fsl_lpi2c.h> LPI2C master data match configuration structure.

Public Members

lpi2c_data_match_config_mode_t matchMode

Data match configuration setting.

bool rxDataMatchOnly

When set to true, received data is ignored until a successful match.

uint32_t match0

Match value 0.

uint32_t match1

Match value 1.

struct *_lpi2c_master_transfer*

#include <fsl_lpi2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_lpi2c_master_transfer_flags` for available options. Set to 0 or `kLPI2C_TransferDefaultFlag` for normal transfers.

uint16_t slaveAddress

The 7-bit slave address.

lpi2c_direction_t direction

Either `kLPI2C_Read` or `kLPI2C_Write`.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct `_lpi2c_master_handle`

#include <fsl_lpi2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint16_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

uint16_t commandBuffer[6]

LPI2C command sequence. When all 6 command words are used: `Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word]`

lpi2c_master_transfer_t transfer

Copy of the current transfer info.

lpi2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData

Application data passed to callback.

struct hostRequest

Public Members

bool enable

Enable host request.

lpi2c_host_request_source_t source

Host request source.

lpi2c_host_request_polarity_t polarity

Host request pin polarity.

2.35 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.

2.36 LPI2C Slave Driver

```
void LPI2C_SlaveGetDefaultConfig(lpi2c_slave_config_t *slaveConfig)
```

Provides a default configuration for the LPI2C slave peripheral.

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave      = true;
slaveConfig->address0         = 0U;
slaveConfig->address1         = 0U;
slaveConfig->addressMatchMode = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable = true;
slaveConfig->filterEnable     = true;
slaveConfig->enableGeneralCall = false;
slaveConfig->sclStall.enableAck = false;
slaveConfig->sclStall.enableTx  = true;
slaveConfig->sclStall.enableRx  = true;
```

(continues on next page)

(continued from previous page)

```

slaveConfig->sclStall.enableAddress    = true;
slaveConfig->ignoreAck                 = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns   = 0;
slaveConfig->sclGlitchFilterWidth_ns   = 0;
slaveConfig->dataValidDelay_ns         = 0;
slaveConfig->clockHoldTime_ns          = 0;

```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `LPI2C_SlaveInit()`. Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure that is set to default values. Refer to `lpi2c_slave_config_t`.

```
void LPI2C_SlaveInit(LPI2C_Type *base, const lpi2c_slave_config_t *slaveConfig, uint32_t
                    sourceClock_Hz)
```

Initializes the LPI2C slave peripheral.

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The LPI2C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `LPI2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

```
void LPI2C_SlaveDeinit(LPI2C_Type *base)
```

Deinitializes the LPI2C slave peripheral.

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveReset(LPI2C_Type *base)
```

Performs a software reset of the LPI2C slave peripheral.

Parameters

- `base` – The LPI2C peripheral base address.

```
static inline void LPI2C_SlaveEnable(LPI2C_Type *base, bool enable)
```

Enables or disables the LPI2C module as slave.

Parameters

- `base` – The LPI2C peripheral base address.
- `enable` – Pass true to enable or false to disable the specified LPI2C as slave.

```
static inline uint32_t LPI2C_SlaveGetStatusFlags(LPI2C_Type *base)
```

Gets the LPI2C slave status flags.

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_lpi2c_slave_flags`

Parameters

- `base` – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void LPI2C_SlaveClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)
```

Clears the LPI2C status flag state.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

Attempts to clear other flags has no effect.

See also:

`_lpi2c_slave_flags`.

Parameters

- `base` – The LPI2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_lpi2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `LPI2C_SlaveGetStatusFlags()`.

```
static inline void LPI2C_SlaveEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Enables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void LPI2C_SlaveDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)
```

Disables the LPI2C slave interrupt requests.

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Parameters

- `base` – The LPI2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_lpi2c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t LPI2C_SlaveGetEnabledInterrupts(LPI2C_Type *base)
```

Returns the set of currently enabled LPI2C slave interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of `_lpi2c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline void LPI2C_SlaveEnableDMA(LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)
```

Enables or disables the LPI2C slave peripheral DMA requests.

Parameters

- base – The LPI2C peripheral base address.
- enableAddressValid – Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
- enableRx – Enable flag for the receive data DMA request. Pass true for enable, false for disable.
- enableTx – Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

```
static inline bool LPI2C_SlaveGetBusIdleState(LPI2C_Type *base)
```

Returns whether the bus is idle.

Requires the slave mode to be enabled.

Parameters

- base – The LPI2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

```
static inline void LPI2C_SlaveTransmitAck(LPI2C_Type *base, bool ackOrNack)
```

Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.

Use this function to send an ACK or NAK when the `kLPI2C_SlaveTransmitAckFlag` is asserted. This only happens if you enable the `sclStall.enableAck` field of the `lpi2c_slave_config_t` configuration structure used to initialize the slave peripheral.

Parameters

- base – The LPI2C peripheral base address.
- ackOrNack – Pass true for an ACK or false for a NAK.

```
static inline void LPI2C_SlaveEnableAckStall(LPI2C_Type *base, bool enable)
```

Enables or disables ACKSTALL.

When enables ACKSTALL, software can transmit either an ACK or NAK on the I2C bus in response to a byte from the master.

Parameters

- base – The LPI2C peripheral base address.
- enable – True will enable ACKSTALL, false will disable ACKSTALL.

static inline uint32_t LPI2C_SlaveGetReceivedAddress(LPI2C_Type *base)

Returns the slave address sent by the I2C master.

This function should only be called if the kLPI2C_SlaveAddressValidFlag is asserted.

Parameters

- base – The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

status_t LPI2C_SlaveSend(LPI2C_Type *base, void *txBuff, size_t txSize, size_t *actualTxSize)

Performs a polling send transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- actualTxSize – **[out]**

Returns

Error or success status returned by API.

status_t LPI2C_SlaveReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.
- actualRxSize – **[out]**

Returns

Error or success status returned by API.

void LPI2C_SlaveTransferCreateHandle(LPI2C_Type *base, lpi2c_slave_handle_t *handle, lpi2c_slave_transfer_callback_t callback, void *userData)

Creates a new handle for the LPI2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_SlaveTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

```
status_t LPI2C_SlaveTransferNonBlocking(LPI2C_Type *base, lpi2c_slave_handle_t *handle,
                                         uint32_t eventMask)
```

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of *lpi2c_slave_transfer_event_t* enumerators for the events you wish to receive. The *kLPI2C_SlaveTransmitEvent* and *kLPI2C_SlaveReceiveEvent* events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the *kLPI2C_SlaveAllEvents* constant is provided as a convenient way to enable all events.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *lpi2c_slave_handle_t* structure which stores the transfer state.
- *eventMask* – Bit mask formed by OR'ing together *lpi2c_slave_transfer_event_t* enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and *kLPI2C_SlaveAllEvents* to enable all events.

Return values

- *kStatus_Success* – Slave transfers were successfully started.
- *kStatus_LPI2C_Busy* – Slave transfers have already been started on this handle.

```
status_t LPI2C_SlaveTransferGetCount(LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t
                                     *count)
```

Gets the slave transfer status during a non-blocking transfer.

Parameters

- *base* – The LPI2C peripheral base address.
- *handle* – Pointer to *i2c_slave_handle_t* structure.
- *count* – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- *kStatus_Success* –
- *kStatus_NoTransferInProgress* –

```
void LPI2C_SlaveTransferAbort(LPI2C_Type *base, lpi2c_slave_handle_t *handle)
```

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- *base* – The LPI2C peripheral base address.

- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`void LPI2C_SlaveTransferHandleIRQ(LPI2C_Type *base, lpi2c_slave_handle_t *handle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The LPI2C peripheral base address.
- `handle` – Pointer to `lpi2c_slave_handle_t` structure which stores the transfer state.

`enum _lpi2c_slave_flags`

LPI2C slave peripheral flags.

The following status register flags can be cleared:

- `kLPI2C_SlaveRepeatedStartDetectFlag`
- `kLPI2C_SlaveStopDetectFlag`
- `kLPI2C_SlaveBitErrFlag`
- `kLPI2C_SlaveFifoErrFlag`

All flags except `kLPI2C_SlaveBusyFlag` and `kLPI2C_SlaveBusBusyFlag` can be enabled as interrupts.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator `kLPI2C_SlaveTxReadyFlag`

Transmit data flag

enumerator `kLPI2C_SlaveRxReadyFlag`

Receive data flag

enumerator `kLPI2C_SlaveAddressValidFlag`

Address valid flag

enumerator `kLPI2C_SlaveTransmitAckFlag`

Transmit ACK flag

enumerator `kLPI2C_SlaveRepeatedStartDetectFlag`

Repeated start detect flag

enumerator `kLPI2C_SlaveStopDetectFlag`

Stop detect flag

enumerator `kLPI2C_SlaveBitErrFlag`

Bit error flag

enumerator `kLPI2C_SlaveFifoErrFlag`

FIFO error flag

enumerator `kLPI2C_SlaveAddressMatch0Flag`

Address match 0 flag

enumerator kLPI2C_SlaveAddressMatch1Flag

Address match 1 flag

enumerator kLPI2C_SlaveGeneralCallFlag

General call flag

enumerator kLPI2C_SlaveBusyFlag

Master busy flag

enumerator kLPI2C_SlaveBusBusyFlag

Bus busy flag

enumerator kLPI2C_SlaveClearFlags

All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_SlaveIrqFlags

IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_SlaveErrorFlags

Errors to check for.

enum _lpi2c_slave_address_match

LPI2C slave address match options.

Values:

enumerator kLPI2C_MatchAddress0

Match only address 0.

enumerator kLPI2C_MatchAddress0OrAddress1

Match either address 0 or address 1.

enumerator kLPI2C_MatchAddress0ThroughAddress1

Match a range of slave addresses from address 0 through address 1.

enum _lpi2c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kLPI2C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kLPI2C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kLPI2C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kLPI2C_SlaveTransmitAckEvent

Callback needs to either transmit an ACK or NACK.

enumerator kLPI2C_SlaveRepeatedStartEvent

A repeated start was detected.

enumerator kLPI2C_SlaveCompletionEvent

A stop was detected, completing the transfer.

enumerator kLPI2C_SlaveAllEvents

Bit mask of all available events.

typedef enum *_lpi2c_slave_address_match* lpi2c_slave_address_match_t

LPI2C slave address match options.

typedef struct *_lpi2c_slave_config* lpi2c_slave_config_t

Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_lpi2c_slave_transfer_event* lpi2c_slave_transfer_event_t

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct *_lpi2c_slave_transfer* lpi2c_slave_transfer_t

LPI2C slave transfer structure.

typedef struct *_lpi2c_slave_handle* lpi2c_slave_handle_t

LPI2C slave handle structure.

typedef void (*lpi2c_slave_transfer_callback_t)(LPI2C_Type *base, *lpi2c_slave_transfer_t* *transfer, void *userData)

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the LPI2C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

struct *_lpi2c_slave_config*

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

`bool enableSlave`
Enable slave mode.

`uint8_t address0`
Slave's 7-bit address.

`uint8_t address1`
Alternate slave 7-bit address.

`lpi2c_slave_address_match_t addressMatchMode`
Address matching options.

`bool filterDozeEnable`
Enable digital glitch filter in doze mode.

`bool filterEnable`
Enable digital glitch filter.

`bool enableGeneralCall`
Enable general call address matching.

`struct _lpi2c_slave_config sclStall`
SCL stall enable options.

`bool ignoreAck`
Continue transfers after a NACK is detected.

`bool enableReceivedAddressRead`
Enable reading the address received address as the first byte of data.

`uint32_t sdaGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SDA signal. Set to 0 to disable.

`uint32_t sclGlitchFilterWidth_ns`
Width in nanoseconds of the digital filter on the SCL signal. Set to 0 to disable.

`uint32_t dataValidDelay_ns`
Width in nanoseconds of the data valid delay.

`uint32_t clockHoldTime_ns`
Width in nanoseconds of the clock hold time.

`struct _lpi2c_slave_transfer`
#include <fsl_lpi2c.h> LPI2C slave transfer structure.

Public Members

`lpi2c_slave_transfer_event_t event`
Reason the callback is being invoked.

`uint8_t receivedAddress`
Matching address send by master.

`uint8_t *data`
Transfer buffer

`size_t dataSize`
Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kLPI2C_SlaveCompletionEvent.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct *_lpi2c_slave_handle*

#include <fsl_lpi2c.h> LPI2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

lpi2c_slave_transfer_t transfer

LPI2C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

lpi2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

struct sclStall

Public Members

bool enableAck

Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted. Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

bool enableTx

Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.

bool enableRx

Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.

bool enableAddress

Enables SCL clock stretching when the address valid flag is asserted.

2.37 LPIT: Low-Power Interrupt Timer

`void LPIT_Init(LPIT_Type *base, const lpit_config_t *config)`

Ungates the LPIT clock and configures the peripheral for a basic operation.

This function issues a software reset to reset all channels and registers except the Module Control register.

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

Parameters

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

`void LPIT_Deinit(LPIT_Type *base)`

Disables the module and gates the LPIT clock.

Parameters

- `base` – LPIT peripheral base address.

`void LPIT_GetDefaultConfig(lpit_config_t *config)`

Fills in the LPIT configuration structure with default settings.

The default values are:

```
config->enableRunInDebug = false;
config->enableRunInDoze = false;
```

Parameters

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

`status_t LPIT_SetupChannel(LPIT_Type *base, lpit_chnl_t channel, const lpit_chnl_params_t *chnlSetup)`

Sets up an LPIT channel based on the user's preference.

This function sets up the operation mode to one of the options available in the enumeration `lpit_timer_modes_t`. It sets the trigger source as either internal or external, trigger selection and the timers behaviour when a timeout occurs. It also chains the timer if a prior timer is requested by the user.

Parameters

- `base` – LPIT peripheral base address.
- `channel` – Channel that is being configured.
- `chnlSetup` – Configuration parameters.

`static inline void LPIT_EnableInterrupts(LPIT_Type *base, uint32_t mask)`

Enables the selected PIT interrupts.

Parameters

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

```
static inline void LPIT_DisableInterrupts(LPIT_Type *base, uint32_t mask)
```

Disables the selected PIT interrupts.

Parameters

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

```
static inline uint32_t LPIT_GetEnabledInterrupts(LPIT_Type *base)
```

Gets the enabled LPIT interrupts.

Parameters

- base – LPIT peripheral base address.

Returns

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

```
static inline uint32_t LPIT_GetStatusFlags(LPIT_Type *base)
```

Gets the LPIT status flags.

Parameters

- base – LPIT peripheral base address.

Returns

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

```
static inline void LPIT_ClearStatusFlags(LPIT_Type *base, uint32_t mask)
```

Clears the LPIT status flags.

Parameters

- base – LPIT peripheral base address.
- mask – The status flags to clear. This is a logical OR of members of the enumeration `lpit_status_flags_t`

```
static inline void LPIT_SetTimerPeriod(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

Timers begin counting down from the value set by this function until it reaches 0, at which point it generates an interrupt and loads this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

```
static inline void LPIT_SetTimerValue(LPIT_Type *base, lpit_chnl_t channel, uint32_t ticks)
```

Sets the timer period in units of count.

In the Dual 16-bit Periodic Counter mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.

Note: Set TVAL register to 0 or 1 is invalid in compare mode.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

static inline uint32_t LPIT_GetCurrentTimerCount(LPIT_Type *base, *lpit_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to microseconds or milliseconds.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

Returns

Current timer counting value in ticks.

static inline void LPIT_StartTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Starts the timer counting.

After calling this function, timers load the period value and count down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

static inline void LPIT_StopTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Stops the timer counting.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

FSL_LPIT_DRIVER_VERSION

Version 2.1.3

enum *_lpit_chnl*

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

Values:

enumerator `kLPIT_Chnl_0`

LPIT channel number 0

enumerator kLPIT_Chnl_1
LPIT channel number 1

enumerator kLPIT_Chnl_2
LPIT channel number 2

enumerator kLPIT_Chnl_3
LPIT channel number 3

enum _lpit_timer_modes

Mode options available for the LPIT timer.

Values:

enumerator kLPIT_PeriodicCounter
Use the all 32-bits, counter loads and decrements to zero

enumerator kLPIT_DualPeriodicCounter
Counter loads, lower 16-bits decrement to zero, then upper 16-bits decrement

enumerator kLPIT_TriggerAccumulator
Counter loads on first trigger and decrements on each trigger

enumerator kLPIT_InputCapture
Counter loads with 0xFFFFFFFF, decrements to zero. It stores the inverse of the current value when a input trigger is detected

enum _lpit_trigger_select

Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

Values:

enumerator kLPIT_Trigger_TimerChn0
Channel 0 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn1
Channel 1 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn2
Channel 2 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn3
Channel 3 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn4
Channel 4 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn5
Channel 5 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn6
Channel 6 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn7
Channel 7 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn8
Channel 8 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn9
Channel 9 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn10

Channel 10 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn11

Channel 11 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn12

Channel 12 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn13

Channel 13 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn14

Channel 14 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn15

Channel 15 is selected as a trigger source

enum _lpit_trigger_source

Trigger source options available.

Values:

enumerator kLPIT_TriggerSource_External

Use external trigger input

enumerator kLPIT_TriggerSource_Internal

Use internal trigger

enum _lpit_interrupt_enable

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerInterruptEnable

Channel 0 Timer interrupt

enumerator kLPIT_Channel1TimerInterruptEnable

Channel 1 Timer interrupt

enumerator kLPIT_Channel2TimerInterruptEnable

Channel 2 Timer interrupt

enumerator kLPIT_Channel3TimerInterruptEnable

Channel 3 Timer interrupt

enum _lpit_status_flags

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerFlag

Channel 0 Timer interrupt flag

enumerator kLPIT_Channel1TimerFlag

Channel 1 Timer interrupt flag

enumerator kLPIT_Channel2TimerFlag

Channel 2 Timer interrupt flag

enumerator kLPIT_Channel3TimerFlag

Channel 3 Timer interrupt flag

typedef enum *_lpit_chnl* lpit_chnl_t

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

typedef enum *_lpit_timer_modes* lpit_timer_modes_t

Mode options available for the LPIT timer.

typedef enum *_lpit_trigger_select* lpit_trigger_select_t

Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

typedef enum *_lpit_trigger_source* lpit_trigger_source_t

Trigger source options available.

typedef enum *_lpit_interrupt_enable* lpit_interrupt_enable_t

List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef enum *_lpit_status_flags* lpit_status_flags_t

List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef struct *_lpit_chnl_params* lpit_chnl_params_t

Structure to configure the channel timer.

typedef struct *_lpit_config* lpit_config_t

LPIT configuration structure.

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

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

static void LPIT_ResetStateDelay(void)

Short wait for LPIT state reset.

After clear or set LPIT_EN, there should be delay longer than 4 LPIT functional clock.

static inline void LPIT_Reset(LPIT_Type *base)

Performs a software reset on the LPIT module.

This resets all channels and registers except the Module Control Register.

Parameters

- base – LPIT peripheral base address.

LPIT_RESET_STATE_DELAY

Delay used in LPIT_Reset.

The macro value should be larger than $4 * \text{core clock} / \text{LPIT peripheral clock}$.

struct __lpit_chnl_params

#include <fsl_lpit.h> Structure to configure the channel timer.

Public Members

bool chainChannel

true: Timer chained to previous timer; false: Timer not chained

lpit_timer_modes_t timerMode

Timers mode of operation.

lpit_trigger_select_t triggerSelect

Trigger selection for the timer

lpit_trigger_source_t triggerSource

Decides if we use external or internal trigger.

bool enableReloadOnTrigger

true: Timer reloads when a trigger is detected; false: No effect

bool enableStopOnTimeout

true: Timer will stop after timeout; false: does not stop after timeout

bool enableStartOnTrigger

true: Timer starts when a trigger is detected; false: decrement immediately

struct __lpit_config

#include <fsl_lpit.h> LPIT configuration structure.

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

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

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

bool enableRunInDoze

true: Timers run in doze mode; false: Timers stop in doze mode

2.38 LPSPI: Low Power Serial Peripheral Interface

2.39 LPSPI Peripheral driver

void LPSPI_MasterInit(LPSPI_Type *base, const lpspi_master_config_t *masterConfig, uint32_t srcClock_Hz)

Initializes the LPSPI master.

Parameters

- base – LPSPI peripheral address.
- masterConfig – Pointer to structure `lpspi_master_config_t`.
- srcClock_Hz – Module source input clock in Hertz

void LPSPI_MasterGetDefaultConfig(*lpspi_master_config_t* *masterConfig)

Sets the `lpspi_master_config_t` structure to default values.

This API initializes the configuration structure for LPSPI_MasterInit(). The initialized structure can remain unchanged in LPSPI_MasterInit(), or can be modified before calling the LPSPI_MasterInit(). Example:

```
lpspi_master_config_t masterConfig;  
LPSPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – pointer to `lpspi_master_config_t` structure

void LPSPI_SlaveInit(LPSPI_Type *base, const *lpspi_slave_config_t* *slaveConfig)

LPSPI slave configuration.

Parameters

- base – LPSPI peripheral address.
- slaveConfig – Pointer to a structure `lpspi_slave_config_t`.

void LPSPI_SlaveGetDefaultConfig(*lpspi_slave_config_t* *slaveConfig)

Sets the `lpspi_slave_config_t` structure to default values.

This API initializes the configuration structure for LPSPI_SlaveInit(). The initialized structure can remain unchanged in LPSPI_SlaveInit() or can be modified before calling the LPSPI_SlaveInit(). Example:

```
lpspi_slave_config_t slaveConfig;  
LPSPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – pointer to `lpspi_slave_config_t` structure.

void LPSPI_Deinit(LPSPI_Type *base)

De-initializes the LPSPI peripheral. Call this API to disable the LPSPI clock.

Parameters

- base – LPSPI peripheral address.

void LPSPI_Reset(LPSPI_Type *base)

Restores the LPSPI peripheral to reset state. Note that this function sets all registers to reset state. As a result, the LPSPI module can't work after calling this API.

Parameters

- base – LPSPI peripheral address.

uint32_t LPSPI_GetInstance(LPSPI_Type *base)

Get the LPSPI instance from peripheral base address.

Parameters

- base – LPSPI peripheral base address.

Returns

LPSPI instance.

```
static inline void LPSPI_Enable(LPSPI_Type *base, bool enable)
```

Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Parameters

- base – LPSPI peripheral address.
- enable – Pass true to enable module, false to disable module.

```
static inline uint32_t LPSPI_GetStatusFlags(LPSPI_Type *base)
```

Gets the LPSPI status flag state.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI status(in SR register).

```
static inline uint8_t LPSPI_GetTxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Tx FIFO size.

```
static inline uint8_t LPSPI_GetRxFifoSize(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Rx FIFO size.

```
static inline uint32_t LPSPI_GetTxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Tx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the transmit FIFO.

```
static inline uint32_t LPSPI_GetRxFifoCount(LPSPI_Type *base)
```

Gets the LPSPI Rx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the receive FIFO.

```
static inline void LPSPI_ClearStatusFlags(LPSPI_Type *base, uint32_t statusFlags)
```

Clears the LPSPI status flag.

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the `_lpspi_flags`. Example usage:

```
LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag|kLPSPI_RxDataReadyFlag);
```

Parameters

- base – LPSPI peripheral address.
- statusFlags – The status flag used from type `_lpspi_flags`.

static inline uint32_t LPSPI_GetTcr(LPSPI_Type *base)

static inline void LPSPI_EnableInterrupts(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI interrupts.

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_DisableInterrupts(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI interrupts.

```
LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_interrupt_enable`.

static inline void LPSPI_EnableDMA(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

static inline void LPSPI_DisableDMA(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
SPI_DisableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum `_lpspi_dma_enable`.

static inline uint32_t LPSPI_GetTxRegisterAddress(LPSPI_Type *base)

Gets the LPSPI Transmit Data Register address for a DMA operation.

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Transmit Data Register address.

```
static inline uint32_t LPSPI_GetRxRegisterAddress(LPSPI_Type *base)
```

Gets the LPSPI Receive Data Register address for a DMA operation.

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Receive Data Register address.

```
bool LPSPI_CheckTransferArgument(LPSPI_Type *base, lpspi_transfer_t *transfer, bool isEdma)
```

Check the argument for transfer .

Parameters

- base – LPSPI peripheral address.
- transfer – the transfer struct to be used.
- isEdma – True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns

Return true for right and false for wrong.

```
static inline void LPSPI_SetMasterSlaveMode(LPSPI_Type *base, lpspi_master_slave_mode_t mode)
```

Configures the LPSPI for either master or slave.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

- base – LPSPI peripheral address.
- mode – Mode setting (master or slave) of type lpspi_master_slave_mode_t.

```
static inline void LPSPI_SelectTransferPCS(LPSPI_Type *base, lpspi_which_pcs_t select)
```

Configures the peripheral chip select used for the transfer.

Parameters

- base – LPSPI peripheral address.
- select – LPSPI Peripheral Chip Select (PCS) configuration.

```
static inline void LPSPI_SetPCSContinuous(LPSPI_Type *base, bool IsContinuous)
```

Set the PCS signal to continuous or uncontinuous mode.

Note: In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

- base – LPSPI peripheral address.
- IsContinuous – True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

static inline bool LPSPI_IsMaster(LPSPI_Type *base)

Returns whether the LPSPI module is in master mode.

Parameters

- base – LPSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

static inline void LPSPI_FlushFifo(LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)

Flushes the LPSPI FIFOs.

Parameters

- base – LPSPI peripheral address.
- flushTxFifo – Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
- flushRxFifo – Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

static inline void LPSPI_SetFifoWatermarks(LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)

Sets the transmit and receive FIFO watermark values.

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

- base – LPSPI peripheral address.
- txWater – The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
- rxWater – The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPSPI_SetAllPcsPolarity(LPSPI_Type *base, uint32_t mask)

Configures all LPSPI peripheral chip select polarities simultaneously.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow | kLPSPI_Pcs1ActiveLow);
```

Parameters

- base – LPSPI peripheral address.
- mask – The PCS polarity mask; Use the enum `_lpspi_pcs_polarity`.

static inline void LPSPI_SetFrameSize(LPSPI_Type *base, uint32_t frameSize)

Configures the frame size.

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame

size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPI in slave mode, although the command register does not update until after the LPSPI is enabled. After it is enabled, the transmit command register should only be changed if the LPSPI is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

- base – LPSPI peripheral address.
- frameSize – The frame size in number of bits.

```
uint32_t LPSPI_MasterSetBaudRate(LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t
                                srcClock_Hz, uint32_t *tcrPrescaleValue)
```

Sets the LPSPI baud rate in bits per second.

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale tcrPrescaleValue parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- baudRate_Bps – The desired baud rate in bits per second.
- srcClock_Hz – Module source input clock in Hertz.
- tcrPrescaleValue – The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a “0” if the LPSPI is not configured for master mode or if the LPSPI module is not disabled.

```
void LPSPI_MasterSetDelayScaler(LPSPI_Type *base, uint32_t scaler, lpspi_delay_type_t
                                whichDelay)
```

Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type *lpspi_delay_type_t*.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.

- `scaler` – The 8-bit delay value 0x00 to 0xFF (255).
- `whichDelay` – The desired delay to configure, must be of type `lpspi_delay_type_t`.

```
uint32_t LPSPI_MasterSetDelayTimes(LPSPI_Type *base, uint32_t delayTimeInNanoSec,  
                                   lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)
```

Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPi module must be configured for master mode before configuring this. And note that the `delayTime = LPSPI_clockSource / (PRESCALE * Delay_scaler)`.

Parameters

- `base` – LPSPi peripheral address.
- `delayTimeInNanoSec` – The desired delay value in nano-seconds.
- `whichDelay` – The desired delay to configuration, which must be of type `lpspi_delay_type_t`.
- `srcClock_Hz` – Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

```
static inline void LPSPI_WriteData(LPSPI_Type *base, uint32_t data)
```

Writes data into the transmit data buffer.

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

- `base` – LPSPi peripheral address.
- `data` – The data word to be sent.

```
static inline uint32_t LPSPI_ReadData(LPSPI_Type *base)
```

Reads data from the data buffer.

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

- `base` – LPSPi peripheral address.

Returns

The data read from the data buffer.

```
void LPSPI_SetDummyData(LPSPI_Type *base, uint8_t dummyData)
```

Set up the dummy data.

Parameters

- *base* – LPSPI peripheral address.
- *dummyData* – Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

```
void LPSPI_MasterTransferCreateHandle(LPSPI_Type *base, lpspi_master_handle_t *handle,
                                     lpspi_master_transfer_callback_t callback, void
                                     *userData)
```

Initializes the LPSPI master handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – LPSPI handle pointer to *lpspi_master_handle_t*.
- *callback* – DSPI callback.
- *userData* – callback function parameter.

```
status_t LPSPI_MasterTransferBlocking(LPSPI_Type *base, lpspi_transfer_t *transfer)
```

LPSPI master transfer data using a polling method.

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *transfer* – pointer to *lpspi_transfer_t* structure.

Returns

status of *status_t*.

```
status_t LPSPI_MasterTransferNonBlocking(LPSPI_Type *base, lpspi_master_handle_t *handle,
                                         lpspi_transfer_t *transfer)
```

LPSPI master transfer data using an interrupt method.

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- *base* – LPSPI peripheral address.
- *handle* – pointer to *lpspi_master_handle_t* structure which stores the transfer state.

- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_MasterTransferGetCount(LPSPI_Type *base, *lpspi_master_handle_t* *handle, size_t *count)

Gets the master transfer remaining bytes.

This function gets the master transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

void LPSPI_MasterTransferAbort(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI master abort transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

void LPSPI_MasterTransferHandleIRQ(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI Master IRQ handler function.

This function processes the LPSPI transmit and receive IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_master_handle_t` structure which stores the transfer state.

void LPSPI_SlaveTransferCreateHandle(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_slave_transfer_callback_t* callback, void *userData)

Initializes the LPSPI slave handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to `lpspi_slave_handle_t`.
- callback – DSPI callback.
- userData – callback function parameter.

`status_t` LPSPI_SlaveTransferNonBlocking(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfer data using an interrupt method.

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- transfer – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`status_t` LPSPI_SlaveTransferGetCount(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, size_t *count)

Gets the slave transfer remaining bytes.

This function gets the slave transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

void LPSPI_SlaveTransferAbort(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI slave aborts a transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

void LPSPI_SlaveTransferHandleIRQ(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI Slave IRQ handler function.

This function processes the LPSPI transmit and receives an IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to `lpspi_slave_handle_t` structure which stores the transfer state.

bool LPSPI_WaitTxFifoEmpty(LPSPI_Type *base)

Wait for tx FIFO to be empty.

This function wait the tx fifo empty

Parameters

- base – LPSPI peripheral address.

Returns

true for the tx FIFO is ready, false is not.

void LPSPI_DriverIRQHandler(uint32_t instance)

LPSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for LPSPI.

Parameters

- instance – LPSPI instance.

FSL_LPSPI_DRIVER_VERSION

LPSPI driver version.

Status for the LPSPI driver.

Values:

enumerator kStatus_LPSPI_Busy

LPSPI transfer is busy.

enumerator kStatus_LPSPI_Error

LPSPI driver error.

enumerator kStatus_LPSPI_Idle

LPSPI is idle.

enumerator kStatus_LPSPI_OutOfRange

LPSPI transfer out Of range.

enumerator kStatus_LPSPI_Timeout

LPSPI timeout polling status flags.

enum _lpspi_flags

LPSPI status flags in SPIx_SR register.

Values:

enumerator kLPSPI_TxDataRequestFlag

Transmit data flag

enumerator kLPSPI_RxDataReadyFlag

Receive data flag

enumerator kLPSPI_WordCompleteFlag

Word Complete flag

enumerator kLPSPI_FrameCompleteFlag

Frame Complete flag

enumerator kLPSPI_TransferCompleteFlag

Transfer Complete flag

enumerator kLPSPI_TransmitErrorFlag

Transmit Error flag (FIFO underrun)

enumerator kLPSPI_ReceiveErrorFlag

Receive Error flag (FIFO overrun)

enumerator kLPSPI_DataMatchFlag

Data Match flag

enumerator kLPSPI_ModuleBusyFlag

Module Busy flag

enumerator kLPSPI_AllStatusFlag

Used for clearing all w1c status flags

enum _lpspi_interrupt_enable

LPSPI interrupt source.

Values:

enumerator kLPSPI_TxInterruptEnable

Transmit data interrupt enable

enumerator kLPSPI_RxInterruptEnable

Receive data interrupt enable

enumerator kLPSPI_WordCompleteInterruptEnable

Word complete interrupt enable

enumerator kLPSPI_FrameCompleteInterruptEnable

Frame complete interrupt enable

enumerator kLPSPI_TransferCompleteInterruptEnable

Transfer complete interrupt enable

enumerator kLPSPI_TransmitErrorInterruptEnable

Transmit error interrupt enable(FIFO underrun)

enumerator kLPSPI_ReceiveErrorInterruptEnable

Receive Error interrupt enable (FIFO overrun)

enumerator kLPSPI_DataMatchInterruptEnable

Data Match interrupt enable

enumerator kLPSPI_AllInterruptEnable

All above interrupts enable.

enum _lpspi_dma_enable

LPSPI DMA source.

Values:

enumerator kLPSPI_TxDmaEnable

Transmit data DMA enable

enumerator kLPSPI_RxDmaEnable

Receive data DMA enable

enum _lpspi_master_slave_mode

LPSPI master or slave mode configuration.

Values:

enumerator kLPSPI_Master

LPSPI peripheral operates in master mode.

enumerator kLPSPI_Slave

LPSPI peripheral operates in slave mode.

enum _lpspi_which_pcs_config

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

Values:

enumerator kLPSPI_Pcs0

PCS[0]

enumerator kLPSPI_Pcs1
PCS[1]

enumerator kLPSPI_Pcs2
PCS[2]

enumerator kLPSPI_Pcs3
PCS[3]

enum __lpspi_pcs_polarity_config
LPSPI Peripheral Chip Select (PCS) Polarity configuration.
Values:

enumerator kLPSPI_PcsActiveHigh
PCS Active High (idles low)

enumerator kLPSPI_PcsActiveLow
PCS Active Low (idles high)

enum __lpspi_pcs_polarity
LPSPI Peripheral Chip Select (PCS) Polarity.
Values:

enumerator kLPSPI_Pcs0ActiveLow
Pcs0 Active Low (idles high).

enumerator kLPSPI_Pcs1ActiveLow
Pcs1 Active Low (idles high).

enumerator kLPSPI_Pcs2ActiveLow
Pcs2 Active Low (idles high).

enumerator kLPSPI_Pcs3ActiveLow
Pcs3 Active Low (idles high).

enumerator kLPSPI_PcsAllActiveLow
Pcs0 to Pcs5 Active Low (idles high).

enum __lpspi_clock_polarity
LPSPI clock polarity configuration.
Values:

enumerator kLPSPI_ClockPolarityActiveHigh
CPOL=0. Active-high LPSPI clock (idles low)

enumerator kLPSPI_ClockPolarityActiveLow
CPOL=1. Active-low LPSPI clock (idles high)

enum __lpspi_clock_phase
LPSPI clock phase configuration.
Values:

enumerator kLPSPI_ClockPhaseFirstEdge
CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

enumerator kLPSPI_ClockPhaseSecondEdge
CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

enum _lpspi_shift_direction

LPSPi data shifter direction options.

Values:

enumerator kLPSPi_MsbFirst

Data transfers start with most significant bit.

enumerator kLPSPi_LsbFirst

Data transfers start with least significant bit.

enum _lpspi_host_request_select

LPSPi Host Request select configuration.

Values:

enumerator kLPSPi_HostReqExtPin

Host Request is an ext pin.

enumerator kLPSPi_HostReqInternalTrigger

Host Request is an internal trigger.

enum _lpspi_match_config

LPSPi Match configuration options.

Values:

enumerator kLPSPi_MatchDisabled

LPSPi Match Disabled.

enumerator kLPSPi_1stWordEqualsM0orM1

LPSPi Match Enabled.

enumerator kLPSPi_AnyWordEqualsM0orM1

LPSPi Match Enabled.

enumerator kLPSPi_1stWordEqualsM0and2ndWordEqualsM1

LPSPi Match Enabled.

enumerator kLPSPi_AnyWordEqualsM0andNxtWordEqualsM1

LPSPi Match Enabled.

enumerator kLPSPi_1stWordAndM1EqualsM0andM1

LPSPi Match Enabled.

enumerator kLPSPi_AnyWordAndM1EqualsM0andM1

LPSPi Match Enabled.

enum _lpspi_pin_config

LPSPi pin (SDO and SDI) configuration.

Values:

enumerator kLPSPi_SdiInSdoOut

LPSPi SDI input, SDO output.

enumerator kLPSPi_SdiInSdiOut

LPSPi SDI input, SDI output.

enumerator kLPSPi_SdoInSdoOut

LPSPi SDO input, SDO output.

enumerator kLPSPi_SdoInSdiOut

LPSPi SDO input, SDI output.

enum _lpspi_data_out_config

LPSPI data output configuration.

Values:

enumerator kLpspiDataOutRetained

Data out retains last value when chip select is de-asserted

enumerator kLpspiDataOutTristate

Data out is tristated when chip select is de-asserted

enum _lpspi_pcs_function_config

LPSPI cs function configuration.

Values:

enumerator kLPSPi_PcsAsCs

PCS pin select as cs function

enumerator kLPSPi_PcsAsData

PCS pin select as date function

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_MasterWidth1

LPSPi master width shift macro, internal used LPSPi master transfer 1bit

enumerator kLPSPi_MasterWidth2

LPSPi master width shift macro, internal used LPSPi master transfer 2bit

enumerator kLPSPi_MasterWidth4

LPSPi master width shift macro, internal used LPSPi master transfer 4bit

enumerator kLPSPi_MasterPcsContinuous

Is PCS signal continuous

enumerator kLPSPi_MasterByteSwap

Is master swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set `lpspi_shift_direction_t` to MSB).

- i. If you set `bitPerFrame = 8` , no matter the `kLPSPi_MasterByteSwap` flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set `bitPerFrame = 16` : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the `kLPSPi_MasterByteSwap` flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the `kLPSPi_MasterByteSwap` flag.
- iii. If you set `bitPerFrame = 32` : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the `kLPSPi_MasterByteSwap` flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the `kLPSPi_MasterByteSwap` flag.

enum `_lpspi_transfer_config_flag_for_slave`

Use this enumeration for LPSPi slave transfer configFlags.

Values:

enumerator kLPSPi_SlavePcs0

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS0 signal

enumerator kLPSPi_SlavePcs1

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS1 signal

enumerator kLPSPi_SlavePcs2

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS2 signal

enumerator kLPSPi_SlavePcs3

LPSPi slave PCS shift macro , internal used. LPSPi slave transfer use PCS3 signal

enumerator kLPSPi_SlaveByteSwap

Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set `lpspi_shift_direction_t` to MSB).

- i. If you set `bitPerFrame = 8` , no matter the `kLPSPi_SlaveByteSwap` flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
- ii. If you set `bitPerFrame = 16` : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the `kLPSPi_SlaveByteSwap` flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the `kLPSPi_SlaveByteSwap` flag.
- iii. If you set `bitPerFrame = 32` : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the `kLPSPi_SlaveByteSwap` flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the `kLPSPi_SlaveByteSwap` flag.

enum `_lpspi_transfer_state`

LPSPI transfer state, which is used for LPSPI transactional API state machine.

Values:

enumerator `kLPSPI_Idle`

Nothing in the transmitter/receiver.

enumerator `kLPSPI_Busy`

Transfer queue is not finished.

enumerator `kLPSPI_Error`

Transfer error.

typedef enum `_lpspi_master_slave_mode` `lpspi_master_slave_mode_t`

LPSPI master or slave mode configuration.

typedef enum `_lpspi_which_pcs_config` `lpspi_which_pcs_t`

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

typedef enum `_lpspi_pcs_polarity_config` `lpspi_pcs_polarity_config_t`

LPSPI Peripheral Chip Select (PCS) Polarity configuration.

typedef enum `_lpspi_clock_polarity` `lpspi_clock_polarity_t`

LPSPI clock polarity configuration.

typedef enum `_lpspi_clock_phase` `lpspi_clock_phase_t`

LPSPI clock phase configuration.

typedef enum `_lpspi_shift_direction` `lpspi_shift_direction_t`

LPSPI data shifter direction options.

typedef enum `_lpspi_host_request_select` `lpspi_host_request_select_t`

LPSPI Host Request select configuration.

typedef enum `_lpspi_match_config` `lpspi_match_config_t`

LPSPI Match configuration options.

typedef enum `_lpspi_pin_config` `lpspi_pin_config_t`

LPSPI pin (SDO and SDI) configuration.

typedef enum `_lpspi_data_out_config` `lpspi_data_out_config_t`

LPSPI data output configuration.

typedef enum `_lpspi_pcs_function_config` `lpspi_pcs_function_config_t`

LPSPI cs function configuration.

typedef enum `_lpspi_transfer_width` `lpspi_transfer_width_t`

LPSPI transfer width configuration.

typedef enum `_lpspi_delay_type` `lpspi_delay_type_t`

LPSPI delay type selection.

typedef struct `_lpspi_master_config` `lpspi_master_config_t`

LPSPI master configuration structure.

typedef struct `_lpspi_slave_config` `lpspi_slave_config_t`

LPSPI slave configuration structure.

typedef struct `_lpspi_master_handle` `lpspi_master_handle_t`

Forward declaration of the `_lpspi_master_handle` typedefs.

```
typedef struct _lpspi_slave_handle lpspi_slave_handle_t
```

Forward declaration of the *_lpspi_slave_handle* typedefs.

```
typedef void (*lpspi_master_transfer_callback_t)(LPSPI_Type *base, lpspi_master_handle_t *handle, status_t status, void *userData)
```

Master completion callback function pointer type.

Param base

LPSPI peripheral address.

Param handle

Pointer to the handle for the LPSPI master.

Param status

Success or error code describing whether the transfer is completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
typedef void (*lpspi_slave_transfer_callback_t)(LPSPI_Type *base, lpspi_slave_handle_t *handle, status_t status, void *userData)
```

Slave completion callback function pointer type.

Param base

LPSPI peripheral address.

Param handle

Pointer to the handle for the LPSPI slave.

Param status

Success or error code describing whether the transfer is completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
typedef struct _lpspi_transfer lpspi_transfer_t
```

LPSPI master/slave transfer structure.

```
volatile uint8_t g_lpspiDummyData[]
```

Global variable for dummy data value setting.

```
LPSPI_DUMMY_DATA
```

LPSPI dummy data if no Tx data.

Dummy data used for tx if there is not txData.

```
SPI_RETRY_TIMES
```

Retry times for waiting flag.

```
LPSPI_MASTER_PCS_SHIFT
```

LPSPI master PCS shift macro , internal used.

```
LPSPI_MASTER_PCS_MASK
```

LPSPI master PCS shift macro , internal used.

```
LPSPI_MASTER_WIDTH_SHIFT
```

LPSPI master width shift macro, internal used

```
LPSPI_MASTER_WIDTH_MASK
```

LPSPI master width shift mask, 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_pcs_function_config_t pcsFunc

Configures cs pins function.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

bool enableInputDelay

Enable master to sample the input data on a delayed SCK. This can help improve slave setup time. Refer to device data sheet for specific time length.

struct __lpspi_slave_config

#include <fsl_lpspi.h> LPSPI slave configuration structure.

Public Members

uint32_t bitsPerFrame

Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol

Clock polarity.

lpspi_clock_phase_t cpha

Clock phase.

lpspi_shift_direction_t direction

MSB or LSB data shift direction.

lpspi_which_pcs_t whichPcs

Desired Peripheral Chip Select (pcs)

lpspi_pcs_polarity_config_t pcsActiveHighOrLow

Desired PCS active high or low

lpspi_pin_config_t pinCfg

Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig

Configures if the output data is tristated between accesses (LPSPi_PCS is negated).

struct _lpspi_transfer

#include <fsl_lpspi.h> LPSPi master/slave transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData

Receive buffer.

volatile size_t dataSize

Transfer bytes.

uint32_t configFlags

Transfer transfer configuration flags. Set from _lpspi_transfer_config_flag_for_master if the transfer is used for master or _lpspi_transfer_config_flag_for_slave enumeration if the transfer is used for slave.

struct _lpspi_master_handle

#include <fsl_lpspi.h> LPSPi master transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous

Is PCS continuous in transfer.

volatile bool writeTcrInIsr

A flag that whether should write TCR in ISR.

volatile bool isByteSwap

A flag that whether should byte swap.

`volatile bool` `isTxMask`
A flag that whether TCR[TXMSK] is set.

`volatile uint16_t` `bytesPerFrame`
Number of bytes in each frame

`volatile uint16_t` `frameSize`
Backup of TCR[FRAMESZ]

`volatile uint8_t` `fifoSize`
FIFO dataSize.

`volatile uint8_t` `rxWatermark`
Rx watermark.

`volatile uint8_t` `bytesEachWrite`
Bytes for each write TDR.

`volatile uint8_t` `bytesEachRead`
Bytes for each read RDR.

`const uint8_t *volatile` `txData`
Send buffer.

`uint8_t *volatile` `rxData`
Receive buffer.

`volatile size_t` `txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t` `rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t` `writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t` `readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t` `totalByteCount`
Number of transfer bytes

`uint32_t` `txBuffIfNull`
Used if the txData is NULL.

`volatile uint8_t` `state`
LPSPI transfer state , `_lpspi_transfer_state`.

`lpspi_master_transfer_callback_t` `callback`
Completion callback.

`void *``userData`
Callback user data.

`struct` `_lpspi_slave_handle`
`#include <fsl_lpspi.h>` LPSPI slave transfer handle structure used for transactional API.

Public Members

`volatile bool` `isByteSwap`
A flag that whether should byte swap.

`volatile uint8_t fifoSize`
FIFO dataSize.

`volatile uint8_t rxWatermark`
Rx watermark.

`volatile uint8_t bytesEachWrite`
Bytes for each write TDR.

`volatile uint8_t bytesEachRead`
Bytes for each read RDR.

`const uint8_t *volatile txData`
Send buffer.

`uint8_t *volatile rxData`
Receive buffer.

`volatile size_t txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t totalByteCount`
Number of transfer bytes

`volatile uint8_t state`
LPSPI transfer state , `_lpspi_transfer_state`.

`volatile uint32_t errorCount`
Error count for slave transfer.

`lpspi_slave_transfer_callback_t callback`
Completion callback.

`void *userData`
Callback user data.

2.40 LPSPI eDMA Driver

`FSL_LPSPI_EDMA_DRIVER_VERSION`
LPSPI EDMA driver version.

`DMA_MAX_TRANSFER_COUNT`
DMA max transfer size.

`typedef struct _lpspi_master_edma_handle lpspi_master_edma_handle_t`
Forward declaration of the `_lpspi_master_edma_handle` typedefs.

`typedef struct _lpspi_slave_edma_handle lpspi_slave_edma_handle_t`
Forward declaration of the `_lpspi_slave_edma_handle` typedefs.

```
typedef void (*lpspi_master_edma_transfer_callback_t)(LPSPI_Type *base,  
lpspi_master_edma_handle_t *handle, status_t status, void *userData)
```

Completion callback function pointer type.

Param base

LPSPI peripheral base address.

Param handle

Pointer to the handle for the LPSPI master.

Param status

Success or error code describing whether the transfer completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
typedef void (*lpspi_slave_edma_transfer_callback_t)(LPSPI_Type *base,  
lpspi_slave_edma_handle_t *handle, status_t status, void *userData)
```

Completion callback function pointer type.

Param base

LPSPI peripheral base address.

Param handle

Pointer to the handle for the LPSPI slave.

Param status

Success or error code describing whether the transfer completed.

Param userData

Arbitrary pointer-dataSized value passed from the application.

```
void LPSPI_MasterTransferCreateHandleEDMA(LPSPI_Type *base, lpspi_master_edma_handle_t  
*handle, lpspi_master_edma_transfer_callback_t  
callback, void *userData, edma_handle_t  
*edmaRxRegToRxDataHandle, edma_handle_t  
*edmaTxDataToTxRegHandle)
```

Initializes the LPSPI master eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that the LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx are the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Tx DMAMUX source for edmaRxRegToRxDataHandle.

Parameters

- base – LPSPI peripheral base address.
- handle – LPSPI handle pointer to lpspi_master_edma_handle_t.
- callback – LPSPI callback.
- userData – callback function parameter.
- edmaRxRegToRxDataHandle – edmaRxRegToRxDataHandle pointer to edma_handle_t.
- edmaTxDataToTxRegHandle – edmaTxDataToTxRegHandle pointer to edma_handle_t.

status_t LPSPI_MasterTransferEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to *lpspi_master_edma_handle_t* structure which stores the transfer state.
- transfer – pointer to *lpspi_transfer_t* structure.

Returns

status of *status_t*.

status_t LPSPI_MasterTransferPrepareEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, uint32_t configFlags)

LPSPI master config transfer parameter while using eDMA.

This function is preparing to transfer data using eDMA, work with LPSPI_MasterTransferEDMALite.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to *lpspi_master_edma_handle_t* structure which stores the transfer state.
- configFlags – transfer configuration flags. *_lpspi_transfer_config_flag_for_master*.

Return values

- kStatus_Success – Execution successfully.
- kStatus_LPSPI_Busy – The LPSPI device is busy.

Returns

Indicates whether LPSPI master transfer was successful or not.

status_t LPSPI_MasterTransferEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA without configs.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: This API is only for transfer through DMA without configuration. Before calling this API, you must call LPSPI_MasterTransferPrepareEDMALite to configure it once. The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.

- `handle` – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- `transfer` – pointer to `lpspi_transfer_t` structure, `config` field is not used.

Return values

- `kStatus_Success` – Execution successfully.
- `kStatus_LPSPI_Busy` – The LPSPI device is busy.
- `kStatus_InvalidArgument` – The transfer structure is invalid.

Returns

Indicates whether LPSPI master transfer was successful or not.

```
void LPSPI_MasterTransferAbortEDMA(LPSPI_Type *base, lpspi_master_edma_handle_t *handle)
```

LPSPI master aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.

```
status_t LPSPI_MasterTransferGetCountEDMA(LPSPI_Type *base, lpspi_master_edma_handle_t *handle, size_t *count)
```

Gets the master eDMA transfer remaining bytes.

This function gets the master eDMA transfer remaining bytes.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_master_edma_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the EDMA transaction.

Returns

status of `status_t`.

```
void LPSPI_SlaveTransferCreateHandleEDMA(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, lpspi_slave_edma_transfer_callback_t callback, void *userData, edma_handle_t *edmaRxRegToRxDataHandle, edma_handle_t *edmaTxDataToTxRegHandle)
```

Initializes the LPSPI slave eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx as the same source) DMA request source.

(1) For a separated DMA request source, enable and set the Rx DMAMUX source for `edmaRxRegToRxDataHandle` and Tx DMAMUX source for `edmaTxDataToTxRegHandle`. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for `edmaRxRegToRxDataHandle`.

Parameters

- `base` – LPSPI peripheral base address.

- `handle` – LPSPI handle pointer to `lpspi_slave_edma_handle_t`.
- `callback` – LPSPI callback.
- `userData` – callback function parameter.
- `edmaRxRegToRxDataHandle` – `edmaRxRegToRxDataHandle` pointer to `edma_handle_t`.
- `edmaTxDataToTxRegHandle` – `edmaTxDataToTxRegHandle` pointer to `edma_handle_t`.

`status_t` LPSPI_SlaveTransferEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfers data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which return right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of `bytesPerFrame` if `bytesPerFrame` is less than or equal to 4. For `bytesPerFrame` greater than 4: The transfer data size should be equal to `bytesPerFrame` if the `bytesPerFrame` is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of `bytesPerFrame`.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- `transfer` – pointer to `lpspi_transfer_t` structure.

Returns

status of `status_t`.

`void` LPSPI_SlaveTransferAbortEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle)

LPSPI slave aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.

`status_t` LPSPI_SlaveTransferGetCountEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, `size_t` *count)

Gets the slave eDMA transfer remaining bytes.

This function gets the slave eDMA transfer remaining bytes.

Parameters

- `base` – LPSPI peripheral base address.
- `handle` – pointer to `lpspi_slave_edma_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the eDMA transaction.

Returns

status of `status_t`.

`struct` `_lpspi_master_edma_handle`

`#include <fsl_lpspi_edma.h>` LPSPI master eDMA transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous

Is PCS continuous in transfer.

volatile bool isByteSwap

A flag that whether should byte swap.

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

volatile uint8_t bytesLastRead

Bytes for last read RDR.

volatile bool isThereExtraRxBytes

Is there extra RX byte.

const uint8_t *volatile txData

Send buffer.

uint8_t *volatile rxData

Receive buffer.

volatile size_t txRemainingByteCount

Number of bytes remaining to send.

volatile size_t rxRemainingByteCount

Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes

Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes

Read RDR register remaining times.

uint32_t totalByteCount

Number of transfer bytes

edma_tcd_t *lastTimeTCD

Pointer to the lastTime TCD

bool isMultiDMATransmit

Is there multi DMA transmit

volatile uint8_t dmaTransmitTime

DMA Transfer times.

uint32_t lastTimeDataBytes

DMA transmit last Time data Bytes

uint32_t dataBytesEveryTime

Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

edma_transfer_config_t transferConfigRx

Config of DMA rx channel.

edma_transfer_config_t transferConfigTx

Config of DMA tx channel.

uint32_t txBuffIfNull

Used if there is not txData for DMA purpose.

uint32_t rxBuffIfNull

Used if there is not rxData for DMA purpose.

uint32_t transmitCommand

Used to write TCR for DMA purpose.

volatile uint8_t state

LPSPi transfer state , _lpspi_transfer_state.

uint8_t nbytes

eDMA minor byte transfer count initially configured.

lpspi_master_edma_transfer_callback_t callback

Completion callback.

void *userData

Callback user data.

edma_handle_t *edmaRxRegToRxDataHandle

edma_handle_t handle point used for RxReg to RxData buff

edma_handle_t *edmaTxDataToTxRegHandle

edma_handle_t handle point used for TxData to TxReg buff

edma_tcd_t lpspiSoftwareTCD[3]

SoftwareTCD, internal used

struct _lpspi_slave_edma_handle

#include <fsl_lpspi_edma.h> LPSPi slave eDMA transfer handle structure used for transactional API.

Public Members

volatile bool isByteSwap

A flag that whether should byte swap.

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

volatile uint8_t bytesLastRead

Bytes for last read RDR.

`volatile bool` `isThereExtraRxBytes`
Is there extra RX byte.

`uint8_t` `nbytes`
eDMA minor byte transfer count initially configured.

`const uint8_t *volatile` `txData`
Send buffer.

`uint8_t *volatile` `rxData`
Receive buffer.

`volatile size_t` `txRemainingByteCount`
Number of bytes remaining to send.

`volatile size_t` `rxRemainingByteCount`
Number of bytes remaining to receive.

`volatile uint32_t` `writeRegRemainingTimes`
Write TDR register remaining times.

`volatile uint32_t` `readRegRemainingTimes`
Read RDR register remaining times.

`uint32_t` `totalByteCount`
Number of transfer bytes

`uint32_t` `txBuffIfNull`
Used if there is not `txData` for DMA purpose.

`uint32_t` `rxBuffIfNull`
Used if there is not `rxData` for DMA purpose.

`volatile uint8_t` `state`
LPSPI transfer state.

`uint32_t` `errorCount`
Error count for slave transfer.

lpspi_slave_edma_transfer_callback_t `callback`
Completion callback.

`void *``userData`
Callback user data.

*edma_handle_t **`edmaRxRegToRxDataHandle`
edma_handle_t handle point used for RxReg to RxData buff

*edma_handle_t **`edmaTxDataToTxRegHandle`
edma_handle_t handle point used for TxData to TxReg

edma_tcd_t `lpspiSoftwareTCD[2]`
SoftwareTCD, internal used

2.41 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:

enumerator kLPTMR_PrescalerClock_0
Prescaler/glitch filter clock 0 selected.

enumerator kLPTMR_PrescalerClock_1
Prescaler/glitch filter clock 1 selected.

enumerator kLPTMR_PrescalerClock_2
Prescaler/glitch filter clock 2 selected.

enumerator kLPTMR_PrescalerClock_3
Prescaler/glitch filter clock 3 selected.

enum _lptmr_interrupt_enable
List of the LPTMR interrupts.

Values:

enumerator kLPTMR_TimerInterruptEnable

Timer interrupt enable

enum _lptmr_status_flags

List of the LPTMR status flags.

Values:

enumerator kLPTMR_TimerCompareFlag

Timer compare flag

typedef enum _lptmr_pin_select lptmr_pin_select_t

LPTMR pin selection used in pulse counter mode.

typedef enum _lptmr_pin_polarity lptmr_pin_polarity_t

LPTMR pin polarity used in pulse counter mode.

typedef enum _lptmr_timer_mode lptmr_timer_mode_t

LPTMR timer mode selection.

typedef enum _lptmr_prescaler_glitch_value lptmr_prescaler_glitch_value_t

LPTMR prescaler/glitch filter values.

typedef enum _lptmr_prescaler_clock_select lptmr_prescaler_clock_select_t

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

typedef enum _lptmr_interrupt_enable lptmr_interrupt_enable_t

List of the LPTMR interrupts.

typedef enum _lptmr_status_flags lptmr_status_flags_t

List of the LPTMR status flags.

typedef struct _lptmr_config lptmr_config_t

LPTMR config structure.

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

The configuration struct can be made constant so it resides in flash.

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

2.42 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.43 LPUART Driver

status_t LPUART_Init(LPUART_Type *base, const *lpuart_config_t* *config, uint32_t srcClock_Hz)

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefaultConfig() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

status_t LPUART_Deinit(LPUART_Type *base)

Deinitializes a LPUART instance.

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

- base – LPUART peripheral base address.

Return values

- kStatus__Success – Deinit is success.
- kStatus__LPUART_Timeout – Timeout during deinit.

void LPUART_GetDefaultConfig(*lpuart_config_t* *config)

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus__LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus__Success – Set baudrate succeeded.

void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, false to disable.

static inline void LPUART_SetMatchAddress(LPUART_Type *base, uint16_t address1, uint16_t address2)

Set the LPUART address.

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable

bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – LPUART peripheral base address.
- address1 – LPUART slave address1.
- address2 – LPUART slave address2.

```
static inline void LPUART__EnableMatchAddress(LPUART_Type *base, bool match1, bool match2)
```

Enable the LPUART match address feature.

Parameters

- base – LPUART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

```
static inline void LPUART__TransferEnable16Bit(lpuart_handle_t *handle, bool enable)
```

Sets the LPUART using 16bit transmit, only for 9bit or 10bit mode.

This function Enable 16bit Data transmit in *lpuart_handle_t*.

Parameters

- handle – LPUART handle pointer.
- enable – true to enable, false to disable.

```
uint32_t LPUART__GetStatusFlags(LPUART_Type *base)
```

Gets LPUART status flags.

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators *_lpuart_flags*. To check for a specific status, compare the return value with enumerators in the *_lpuart_flags*. For example, to check whether the TX is empty:

```
if (kLPUART__TxDataRegEmptyFlag & LPUART__GetStatusFlags(LPUART1))
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART status flags which are ORed by the enumerators in the *_lpuart_flags*.

```
status_t LPUART__ClearStatusFlags(LPUART_Type *base, uint32_t mask)
```

Clears status flags with a provided mask.

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only be cleared or set by hardware are: `kLPUART_TxDataRegEmptyFlag`, `kLPUART_TransmissionCompleteFlag`, `kLPUART_RxDataRegFullFlag`, `kLPUART_RxActiveFlag`, `kLPUART_NoiseErrorFlag`, `kLPUART_ParityErrorFlag`, `kLPUART_TxFifoEmptyFlag`, `kLPUART_RxFifoEmptyFlag`. Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- `base` – LPUART peripheral base address.
- `mask` – the status flags to be cleared. The user can use the enumerators in the `_lpuart_status_flag_t` to do the OR operation and get the mask.

Return values

- `kStatus_LPUART_FlagCannotClearManually` – The flag can't be cleared by this function but it is cleared automatically by hardware.
- `kStatus_Success` – Status in the mask are cleared.

Returns

0 succeed, others failed.

```
void LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)
```

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the `_lpuart_interrupt_enable`. This example shows how to enable TX empty interrupt and RX full interrupt:

```
LPUART_EnableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – LPUART peripheral base address.
- `mask` – The interrupts to enable. Logical OR of `_lpuart_interrupt_enable`.

```
void LPUART_DisableInterrupts(LPUART_Type *base, uint32_t mask)
```

Disables LPUART interrupts according to a provided mask.

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpuart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
LPUART_DisableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – LPUART peripheral base address.
- `mask` – The interrupts to disable. Logical OR of `_lpuart_interrupt_enable`.

```
uint32_t LPUART_GetEnabledInterrupts(LPUART_Type *base)
```

Gets enabled LPUART interrupts.

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_lpuart_interrupt_enable`. To check a specific interrupt enable status, compare the return value with enumerators in `_lpuart_interrupt_enable`. For example, to check whether the TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);  
  
if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)  
{  
    ...  
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in `_lpuart_interrupt_enable`.

`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.

`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 remainBytes 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

enum _lpuart_stop_bit_count

LPUART stop bit count.

Values:

enumerator kLPUART_OneStopBit

One stop bit

enumerator kLPUART_TwoStopBit

Two stop bits

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_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_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 reg-
    ister. 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_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_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_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__lpuartIRQ[]

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_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 is16bitData

16bit data bits flag, only used for 9bit or 10bit data

union __unnamed41__

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 __unnamed43__

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 __unnamed45__

Public Members

uint8_t *volatile rxData

Address of remaining data to receive.

uint16_t *volatile rxData16

Address of remaining data to receive.

union __unnamed47__

Public Members

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`uint16_t *rxRingBuffer16`
Start address of the receiver ring buffer.

2.44 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

2.45 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.

2.46 Mipi_dsi

`void DSI_Init(MIPI_DSI_Type *base, dsi_config_t *config)`

Initializes the MIPI DSI host with the user configuration.

This function initializes the MIPI DSI host with the configuration, it should be called before other MIPI DSI driver functions.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to the user configuration structure.

`void DSI_Deinit(MIPI_DSI_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)`

Gets the default configuration to initialize the MIPI DSI host.

The default value is:

```
config->mode = kDSI_CommandMode;
config->packageFlags = kDSI_DpiEnableAll;
config->enableNoncontinuousClk = true;
config->HsRxDeviceReady_ByteClk = 0U;
config->lpRxDeviceReady_ByteClk = 0U;
config->HsTxDeviceReady_ByteClk = 0U;
config->lpTxDeviceReady_ByteClk = 0U;
```

Parameters

- config – Pointer to a user-defined configuration structure.

`uint32_t DSI_DphyGetPlldivider(uint32_t *m, uint32_t *n, uint32_t refClkFreq_Hz, uint32_t desiredOutFreq_Hz)`

Calculates the D-PHY PLL dividers to generate the desired output frequency.

The phy byte clock frequency(byte count per second) is generated by multiplying the refClkFreq_Hz, the formula is as follows, m & n is configured by mediamix control block.

$\text{desiredOutFreq\_Hz} = \text{refClkFreq\_Hz} * (M + 2) / (N + 1)$. M: 62 ~ 625 N: 0 ~ 15

Parameters

- m – Control of the feedback multiplication ratio.
- n – Control of the input frequency division ratio.
- refClkFreq_Hz – The D-PHY input reference clock frequency (REF_CLK).
- desiredOutFreq_Hz – Desired PLL output frequency.

Returns

The actually output frequency using the returned dividers. If can not find suitable dividers, return 0.

`status_t DSI_PowerUp(MIPI_DSI_Type *base)`

Power up the DSI.

Parameters

- base – MIPI DSI host peripheral base address.

Return values

- `kStatus_Success` – Data transfer finished with no error.
- `kStatus_Timeout` – Transfer failed because of timeout.

```
static inline void DSI_PowerDown(MIPI_DSI_Type *base)
```

Power down the DSI.

Parameters

- `base` – MIPI DSI host peripheral base address.

```
static inline void DSI_EnableInterrupts(MIPI_DSI_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_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_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.

```
void DSI_SetDpiConfig(MIPI_DSI_Type *base, const dsi_dpi_config_t *config, uint8_t laneNum)
```

Configure the DPI interface.

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.
- `laneNum` – How many lanes in use.

```
void DSI_SetCommandModeConfig(MIPI_DSI_Type *base, const dsi_command_config_t *config,  
uint32_t phyByteClkFreq_Hz)
```

Configures the command mode configuration.

This function configures the timeout values for DSI command mode.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to the command mode configuration structure.
- phyByteClkFreq_Hz – Byte clock frequency in each lane.

static inline void DSI_EnableCommandMode(MIPI_DSI_Type *base, bool enable)

Enables the command mode.

This function configures the timeout values for DSI command mode.

Parameters

- base – MIPI DSI host peripheral base address.
- enable – true to enable command mode and disable video mode, vise versa.

static inline void DSI_EnableVpgEnMode(MIPI_DSI_Type *base, bool enable)

Enables the VPG mode.

This function configures video mode pattern generator.

Parameters

- base – MIPI DSI host peripheral base address.
- enable – true to enable video mode pattern generator.

void DSI_GetDefaultDphyConfig(*dsi_dphy_config_t* *config, uint32_t phyByteClkFreq_Hz, uint8_t laneNum)

Gets the default D-PHY configuration.

Gets the default D-PHY configuration, the timing parameters are set according to D-PHY specification. User can use the configuration directly, or change the parameters according device specific requirements.

Parameters

- config – Pointer to the D-PHY configuration.
- phyByteClkFreq_Hz – Byte clock frequency in each lane.
- laneNum – How may lanes in use.

void DSI_InitDphy(MIPI_DSI_Type *base, const *dsi_dphy_config_t* *config)

Initializes the D-PHY.

This function configures the D-PHY timing and setups the D-PHY PLL based on user configuration. The default configuration can be obtained by calling the function DSI_GetDefaultDphyConfig.

Parameters

- base – MIPI DSI host peripheral base address.
- config – Pointer to the D-PHY configuration.

void DSI_SetPacketControl(MIPI_DSI_Type *base, uint8_t flags)

Configures the APB packet to send.

This function configures the next APB packet transfer feature. After configuration, user can write the payload by calling DSI_WriteTxPayload then call DSI_WriteTxHeader to start the tranasfer or just call DSI_WriteTxHeader alone if it is a short packet.

Parameters

- base – MIPI DSI host peripheral base address.
- flags – The transfer control flags, see ref `_dsi_transfer_flags`.


```
void DSI_WriteTxHeader(MIPI_DSI_Type *base, uint16_t wordCount, uint8_t virtualChannel,
                     dsi_tx_data_type_t dataType)
```

Writes tx header to command FIFO. This will trigger the packet 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).

```
void DSI_WriteTxPayload(MIPI_DSI_Type *base, const uint8_t *payload, uint16_t payloadSize)
```

Fills 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_WriteTxPayloadExt(MIPI_DSI_Type *base, const uint8_t *payload, uint16_t
                          payloadSize, bool sendDcsCmd, uint8_t dcsCmd)
```

Writes payload data to generic payload FIFO.

Write the long packet payload to TX FIFO. This function could be used in two ways

- Include the DCS command in the 1st byte of `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_WriteTxPayload` when used in this way.
- The DCS command is not in `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_ReadRxData(MIPI_DSI_Type *base, uint8_t *payload, uint16_t payloadSize)
```

Reads the long APB packet payload.

Read the long packet payload from RX FIFO. This function reads directly from RX FIFO status. Upper layer should make sure the whole rx packet has been received.

Parameters

- `base` – MIPI DSI host peripheral base address.
- `payload` – Pointer to the payload buffer.
- `payloadSize` – Payload size in byte.

status_t DSI_TransferBlocking(MIPI_DSI_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.

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 ref DSI_GetRxErrorStatus to check the error details.
- kStatus_DSI_PhyError – PHY error detected during transfer.
- kStatus_DSI_ErrorReportReceived – Error Report packet received, user could use ref DSI_GetAndClearHostStatus to check the error report status.
- kStatus_DSI_NotSupported – Transfer format not supported.
- kStatus_DSI_Fail – Transfer failed for other reasons.

uint16_t Pll_Set_Hs_Freqrange(*uint32_t* bnd_width)

Lookup table method to obtain HS frequency range of operation selection override.

Parameters

- bnd_width – band width frequency in Hz

Returns

the *hsfreqrange_ovr*[6:0] value based on band width frequency in hz.

uint16_t Pll_Set_Pll_Prop_Param(*uint32_t* pll_freq_sel)

Lookup table method to obtain PLL Proportional Charge Pump control.

Parameters

- pll_freq_sel – PLL frequency in Mhz

Returns

the *pll_prop_cntrl_rw*[5:0] value based on video Pll frequency in Mhz.

uint16_t Pll_Set_Sr_Osc_Freq_Target(*uint32_t* pll_freq_sel)

Lookup table method to obtain DDL target oscillation frequency.

Parameters

- pll_freq_sel – PLL frequency in Mhz

Returns

the *sr_osc_freq_target*[11:0] value based on video Pll frequency in Mhz.

uint32_t Pll_Set_Pll_Vco_Freq(*uint32_t* pll_freq_sel)

calculate VCO frequency.

Parameters

- pll_freq_sel – PLL frequency in Hz

Returns

the *vco_freq_clk* value

uint16_t Pll_Set_Pll_Vco_Param(uint32_t pll_freq_sel)

Lookup table method to obtain VCO parameter.

Parameters

- pll_freq_sel – PLL frequency in Mhz

Returns

the pll_vco_cntrl_ovr_rw[5:0] value based on video Pll frequency in Mhz. If can not find suitable value, return default value 63.

void DSI_ConfigDphy(MIPI_DSI_Type *base, uint32_t phyRefClkFreq_Hz, uint32_t dataRateFreq_Hz)

config to set Dphy.

Parameters

- phyRefClkFreq_Hz – Dphy reference clock frequency in Hz
- dataRateFreq_Hz – line rate clock frequency.

FSL_MIPI_DSI_DRIVER_VERSION

Error codes for the MIPI DSI driver.

Values:

enumerator kStatus_DSI_Busy

DSI is busy.

enumerator kStatus_DSI_EccMultiBitError

Multibit ECC error detected in rx packet.

enumerator kStatus_DSI_CrcError

CRC error detected in rx packet.

enumerator kStatus_DSI_PacketSizeError

Rx packet size error.

enumerator kStatus_DSI_EotMissingError

Received transmission does not end with an EoT packet.

enumerator kStatus_DSI_ErrorReportReceived

Error report package received.

enumerator kStatus_DSI_NotSupported

The transfer type not supported.

enumerator kStatus_DSI_PhyError

Physical layer error.

Status and interrupt mask of acknowledge error sent by device caused by host, belongs to interrupt group1. INT_ST0 bit0-bit15.

Values:

enumerator kDSI_ErrorReportSot

SoT error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportSotSync

SoT Sync error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportEotSync

EoT Sync error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportEscEntryCmd

Escape Mode Entry Command error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportLpSync

Low-power Transmit Sync error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportPeriphTo

Peripheral Timeout error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportFalseControl

False Control error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportDeviceSpecific1

The deice specific error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportEccOneBit

Single-bit ECC error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportEccMultiBit

Muiti-bit ECC error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportChecksum

Checksum error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportDataTypeUnrecognized

DSI data type not recognized error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportVcIdInvalid

Virtual channel ID invalid error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportTxLengthInvalid

Invalid transmission length error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportDeviceSpecific2

The deice specific error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportProtocolViolation

Protocol violation error detected in device's acknowledge error report.

enumerator kDSI_ErrorReportAll

Status and interrupt mask of error in phy layer, belongs to interrupt group1. INT_ST0 bit16-bit20.

Values:

enumerator kDSI_PhyErrorEscEntry

Escape entry error from Lane 0.

enumerator kDSI_PhyErrorLpSync

Low-power data transmission synchronization error from Lane 0.

enumerator kDSI_PhyErrorControl

Control error from Lane 0.

enumerator kDSI_PhyErrorLp0Connection

LP0 connection error from Lane 0.

enumerator kDSI_PhyErrorLp1Connection

LP1 connection error from Lane 0.

enumerator kDSI_PhyErrorAll

Timeout error interrupt and status, belongs to interrupt group2. INT_ST1 bit0-bit6.

Values:

enumerator kDSI_TimeoutErrorHtx

High Speed forward TX timeout detected.

enumerator kDSI_TimeoutErrorLrx

Reverse Low power data receive timeout detected.

Host receive packet error status, belongs to interrupt group2. INT_ST1 bit0-bit6.

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_RxErrorPacketSize

Packet size error detected.

enumerator kDSI_RxErrorEotMissing

Host receives a transmission that does not end with an EoT packet.

enumerator kDSI_RxErrorAll

Host receive error status, belongs to interrupt group2. INT_ST1 bit7-bit12 bit19?

Values:

enumerator kDSI_DpiPayloadFifoOverflow

During a DPI pixel line storage, the payload FIFO overflow occurs.

enumerator kDSI_GenericCommandFifoOverflow

System writes a command through the Generic interface while FIFO is full causing overflow.

enumerator kDSI_GenericPayloadFifoOverflow

System writes a payload data through the Generic interface while FIFO is full causing payload FIFO overflow.

enumerator kDSI_GenericPayloadFifoUnderflow

System writes the packet header before the packet payload is completed loaded into the payload FIFO during a packet build causing payload FIFO underflow.

enumerator kDSI_GenericReadFifoUnderflow

System requests data before it is fully received causing underflow.

enumerator kDSI_GenericReadFifoOverflow

The Read FIFO size is not correctly dimensioned for the max rx packet size causing generic read FIFO overflow.

`_dsi_dpi_package_flag` Flags for DPI package composition.

Values:

enumerator `kDSI_DpiEnableEotpTxHs`

Enables the EoTp transmission in high-speed.

enumerator `kDSI_DpiEnableEotpRx`

Enables the EoTp reception.

enumerator `kDSI_DpiEnableBta`

Enables the Bus Turn-Around (BTA) request.

enumerator `kDSI_DpiEnableEcc`

Enables the ECC reception, error correction, and reporting.

enumerator `kDSI_DpiEnableCrc`

Enables the CRC reception and error reporting.

enumerator `kDSI_DpiEnableEotpTxLp`

Enables the EoTp transmission in low-power.

enumerator `kDSI_DpiEnableAll`

enum `_dsi_operation_mode`

MIPI DSI operation mode.

Values:

enumerator `kDSI_VideoMode`

Video mode.

enumerator `kDSI_CommandMode`

Command mode.

enum `_dsi_dpi_color_coding`

MIPI DPI interface color coding.

Values:

enumerator `kDSI_DpiRGB16Bit`

16-bit configuration 1. RGB565: XXXXXXXX_RRRRRGGG_GGGBBBBB.

enumerator `kDSI_DpiRGB16BitLoose0`

16-bit configuration 2. RGB565: XXXRRRRR_XXGGGGGG_XXXBBBBB.

enumerator `kDSI_DpiRGB16BitLoose1`

16-bit configuration 3. RGB565: XXRRRRRX_XXGGGGGG_XXBBBBBX.

enumerator `kDSI_DpiRGB18Bit`

18-bit configuration 1. RGB666: XXXXXXRR_RRRRGGGG_GGBBBBBB.

enumerator `kDSI_DpiRGB18BitLoose`

18-bit configuration 2. RGB666: XXRRRRRR_XXGGGGGG_XXBBBBBB.

enumerator `kDSI_DpiRGB24Bit`

24-bit configuration. RGB888: RRRRRRRR_GGGGGGGG_BBBBBBBB.

enumerator `kDSI_DpiYCbCr20Bit`

20-bit configuration. YCbCr422 loosely packed.
Cle1: YYYYYYYY_YYXXCbCbCbCb_CbCbCbCbCbCbXX,
YYYYYYYY_YYXXCrCrCrCr_CrCrCrCrCrCrXX.

Cy-
Cycle2:

enumerator kDSI_DpiYCbCr24Bit

24-bit configuration. YCbCr422: CyCle1: YYYYYYYY_YYYYCbCbCbCb_CbCbCbCbCbCbCbCb,
Cycle2: YYYYYYYY_YYYYCrCrCrCr_CrCrCrCrCrCrCrCr.

enumerator kDSI_DpiYCbCr16Bit

16-bit configuration. YCbCr422 loosely packed. Cy-
Cle1: YYYYYYYY_YYXXCbCbCbCb_CbCbCbCbCbCbXX, Cycle2:
YYYYYYY_XXXXCrCrCrCr_CrCrCrCrXXXX.

enumerator kDSI_DpiRGB30Bit

30-bit configuration. RGB10.10.10: XXRRRRRR_RRRRGGGG_GGGGGGBB_BBBBBBBB.

enumerator kDSI_DpiRGB36Bit

36-bit configuration. RGB12.12.12. CyCle1: XXXXXXRR_RRRRRRRR_RRGGGGGG, Cy-
cle2: XXXXXXGG_GGGGBBBB_BBBBBBBB.

enumerator kDSI_DpiYCbCr12Bit

12-bit configuration. YCbCr420: CyCle1: Y1Y1Y1Y1Y1Y1Y1_Y0Y0Y0Y0Y0Y0Y0_CbCbCbCbCbCbCbCb
Cycle2: Y1Y1Y1Y1Y1Y1Y1Y1_Y0Y0Y0Y0Y0Y0Y0Y0_CrCrCrCrCrCrCrCr.

enumerator kDSI_DpiDcs24Bit

24-bit configuration with no specific coding.

_dsi_dpi_polarity_flag Flags for DPI signal polarity.

Values:

enumerator kDSI_DpiDataEnableActiveHigh

Data enable pin active high.

enumerator kDSI_DpiVsyncActiveHigh

VSYNC active high.

enumerator kDSI_DpiHsyncActiveHigh

HSYNC active high.

enumerator kDSI_DpiShutDownActiveHigh

Shutdown pin active high.

enumerator kDSI_DpiColorModeActiveHigh

Color mode pin active high.

enumerator kDSI_DpiDataEnableActiveLow

Data enable pin active low.

enumerator kDSI_DpiVsyncActiveLow

VSYNC active low.

enumerator kDSI_DpiHsyncActiveLow

HSYNC active low.

enumerator kDSI_DpiShutDownActiveLow

Shutdown pin active low.

enumerator kDSI_DpiColorModeActiveLow

Color mode pin active low.

enum _dsi_video_mode

DSI 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_video_pattern

Values:

enumerator kDSI_PatternDisable
Color bar pattern mode disabled.

enumerator kDSI_PatternVertical
Color bar pattern mode displayed vertically.

enumerator kDSI_PatternHorizontal
Color bar pattern mode displayed horizontally.

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_TransferUseLowPower`

Use low power or not.

enumerator `kDSI_TransferPerformBTA`

Perform BTA or not at the end of a frame.

typedef enum `_dsi_operation_mode` `dsi_operation_mode_t`

MIPI DSI operation mode.

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_video_mode` `dsi_video_mode_t`

DSI video mode.

typedef enum `_dsi_video_pattern` `dsi_video_pattern_t`

typedef struct `_dsi_dpi_config` `dsi_dpi_config_t`

MIPI DSI controller DPI interface configuration.

typedef struct `_dsi_command_config` `dsi_command_config_t`

MIPI DSI command mode 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.

```
uint32_t DSI_GetInstance(MIPI_DSI_Type *base)
```

Gets the MIPI DSI host controller instance from peripheral base address.

Parameters

- base – MIPI DSI peripheral base address.

Returns

MIPI DSI instance.

TX_DPHY_TX_PLL_1

TX_DPHY_TX_PLL_5

TX_DPHY_TX_PLL_9

TX_DPHY_TX_PLL_13

TX_DPHY_TX_PLL_17

TX_DPHY_TX_PLL_22

TX_DPHY_TX_PLL_23

TX_DPHY_TX_PLL_24

TX_DPHY_TX_PLL_25

TX_DPHY_TX_PLL_27

TX_DPHY_TX_PLL_28

TX_DPHY_TX_PLL_29

TX_DPHY_TX_PLL_30

TX_DPHY_TX_PLL_31

TX_DPHY_TX_CB_0

TX_DPHY_TX_CB_1

TX_DPHY_TX_CB_2

TX_DPHY_TX_SLEW_5

TX_DPHY_TX_SLEW_6

TX_DPHY_TX_SLEW_7

TX_DPHY_TX_CLK_TERMLOWCAP

```
struct _dsi_config
```

#include <fsl_mipi_dsi.h> MIPI DSI controller configuration.

Public Members

dsi_operation_mode_t mode

DSI operation mode. MODE_CFG[cmd_video_mode]

uint8_t packageFlags

OR'ed value of _dsi_dpi_package_flag that controls DPI package composition. PCK-HDL_CFG

bool enableNoncontinuousClk

Enables the automatic mechanism to stop providing clock in the clock lane when time allows. LPCLK_CTRL[auto_clklane_ctrl]

uint16_t HsRxDeviceReady_ByteClk

The min time the display device takes to process high-speed read from master before it can continue doing other stuff. The timer starts when D-PHY enters stop state and measured in lane byte clock. HS_RD_TO_CNT[hs_rd_to_cnt]

uint16_t lpRxDeviceReady_ByteClk

The min time the display device takes to process low-power read from master before it can continue doing other stuff. The timer starts when D-PHY enters stop state and measured in lane byte clock. LP_RD_TO_CNT[lp_rd_to_cnt]

uint16_t HsTxDeviceReady_ByteClk

The min time the display device takes to process high-speed write from master before it can continue doing other stuff. The timer starts when D-PHY enters stop state and measured in lane byte clock. HS_WR_TO_CNT[hs_wr_to_cnt]

uint16_t lpTxDeviceReady_ByteClk

The min time the display device takes to process low-power write from master before it can continue doing other stuff. The timer starts when D-PHY enters stop state and measured in lane byte clock. LP_WR_TO_CNT[lp_wr_to_cnt]

struct _dsi_dpi_config

#include <fsl_mipi_dsi.h> MIPI DSI controller DPI interface configuration.

Public Members

uint8_t virtualChannel

Virtual channel. DPI_VCID[dpi_vcid]

dsi_dpi_color_coding_t colorCoding

DPI color coding. DPI_COLOR_CODING

uint8_t polarityFlags

OR'ed value of _dsi_dpi_polarity_flag that controls signal polarity. DPI_CFG_POL

bool enablelpSwitch

Enable return to low-power inside the VSA/VBP/VFP/VACT/HBP/HFP period when timing allows. VID_MODE_CFG[bit8-13]

bool enableAck

Enable the request for an acknowledge response at the end of a frame. VID_MODE_CFG[frame_bta_ack_en]

dsi_video_mode_t videoMode

Video mode. VID_MODE_CFG[vid_mode_type]

uint16_t pixelPayloadSize

Color bar pattern. VID_MODE_CFG[vpg_orientation][vpg_en][vpg_mode=0] The number of pixels in a single video packet. For 18-bit not loosely packed data types, this number must be a multiple of 4, for YCbCr data types, it must be a multiple of 2. Recommended to set to the line size (in pixels). VID_PKT_SIZE

uint16_t vsw

Number of lines in vertical sync width. VID_VSA_LINES

uint16_t vbp

Number of lines in vertical back porch. VID_VBP_LINES

uint16_t vfp

Number of lines in vertical front porch. VID_VFP_LINES

uint16_t panelHeight

Number of lines in vertical active area. VID_VACTIVE_LINES

uint16_t hsw

Horizontal sync width, in dpi pixel clock. VID_HSA_TIME

uint16_t hbp

Horizontal back porch, in dpi pixel clock. VID_HBP_TIME

uint16_t hfp

Horizontal front porch, in dpi pixel clock. VID_HLINE_TIME = (hsw+hbp+hfp+width)

struct __dsi_command_config

#include <fsl_mipi_dsi.h> MIPI DSI command mode configuration.

Public Members

uint32_t escClkFreq_Hz

Escape clock frequency in Hz.

uint16_t lpRxTo_Ns

Timeout value that triggers a low-power reception timeout contention detection. TO_CNT_CFG[lprx_to_cnt]

uint16_t hsTxTo_Ns

Timeout value that triggers a high-speed transmission timeout contention detection. In non-burst mode, the time should be larger than 1.1 times of one frame data transmission time, in burst mode it should be one line. TO_CNT_CFG[hstx_to_cnt]

uint16_t btaTo_Ns

The time period for which MIPI DSI host keeps the link still after completing a Bus Turnaround. BTA_TO_CNT[bta_to_cnt]

struct __dsi_dphy_config

#include <fsl_mipi_dsi.h> MIPI DSI D-PHY configuration.

Public Members

uint8_t numLanes

Number of lanes. The value range is from 1-4, lane 0-3. PHY_IF_CFG[n_lanes]

uint8_t tStopState_ByteClk

Minimum time that the PHY controller stays in stop state before a HS transmission. TODO in what unit? PHY_IF_CFG[phy_stop_wait_time]

uint16_t tClkHs2Lp_ByteClk

Maximum time that the D-PHY clock lane takes to go from high-speed to low-power in lane byte clock. PHY_TMR_LPCLK_CFG[phy_clkhs2lp_time]

uint16_t tClkLp2Hs_ByteClk

Maximum time that the D-PHY clock lane takes to go from low-power to high-speed in lane byte clock. PHY_TMR_LPCLK_CFG[phy_clklp2hs_time]

uint16_t tDataHs2Lp_ByteClk

Maximum time that the D-PHY data lane takes to go from high-speed to low-power in lane byte clock. PHY_TMR_CFG[phy_hs2lp_time]

uint16_t tDataLp2Hs_ByteClk

Maximum time that the D-PHY data lane takes to go from low-power to high-speed in lane byte clock. PHY_TMR_CFG[phy_lp2hs_time]

uint16_t maxRead_ByteClk

Maximum time required to perform a read command in lane byte clock. PHY_TMR_RD_CFG[max_rd_time]

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 TX data buffer.

uint16_t txDataSize

Size of the TX data.

uint16_t rxDataSize

Size of the RX data.

bool sendDcsCmd

If set to true, the DCS command is specified by `dcsCmd`, otherwise the DCS command is included in the `txData`.

uint8_t dcsCmd

The DCS command to send, only valid when `sendDcsCmd` is true.

2.47 MIPI_DSI: MIPI DSI Host Controller

2.48 MU: Messaging Unit

```
uint32_t MU_GetInstance(MU_Type *base)
```

Get the MU instance index.

Parameters

- base – MU peripheral base address.

Returns

MU instance index.

```
void MU_Init(MU_Type *base)
```

Initializes the MU module.

This function enables the MU clock only.

Parameters

- base – MU peripheral base address.

```
void MU_Deinit(MU_Type *base)
```

De-initializes the MU module.

This function disables the MU clock only.

Parameters

- base – MU peripheral base address.

```
static inline void MU_SendMsgNonBlocking(MU_Type *base, uint32_t regIndex, uint32_t msg)
```

Writes a message to the TX register.

This function writes a message to the specific TX register. It does not check whether the TX register is empty or not. The upper layer should make sure the TX register is empty before calling this function. This function can be used in ISR for better performance.

```
while (!(kMU_Tx0EmptyFlag & MU_GetStatusFlags(base))) { } Wait for TX0 register empty.
MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG_VAL); Write message to the TX0 register.
```

Parameters

- base – MU peripheral base address.
- regIndex – TX register index, see `mu_msg_reg_index_t`.
- msg – Message to send.

```
status_t MU_SendMsg(MU_Type *base, uint32_t regIndex, uint32_t msg)
```

Blocks to send a message.

This function waits until the TX register is empty and sends the message. If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and returns `kStatus_Timeout`.

Parameters

- base – MU peripheral base address.
- regIndex – MU message register, see `mu_msg_reg_index_t`.
- msg – Message to send.

Return values

- `kStatus_Success` – Message sent successfully.
- `kStatus_Timeout` – Timeout occurred while waiting for TX register to be empty.

Returns

`status_t`

static inline uint32_t MU_ReceiveMsgNonBlocking(MU_Type *base, uint32_t regIndex)

Reads a message from the RX register.

This function reads a message from the specific RX register. It does not check whether the RX register is full or not. The upper layer should make sure the RX register is full before calling this function. This function can be used in ISR for better performance.

```
uint32_t msg;
while (!(kMU_Rx0FullFlag & MU_GetStatusFlags(base)))
{
    } Wait for the RX0 register full.

msg = MU_ReceiveMsgNonBlocking(base, kMU_MsgReg0); Read message from RX0 register.
```

Parameters

- base – MU peripheral base address.
- regIndex – RX register index, see mu_msg_reg_index_t.

Returns

The received message.

status_t MU_ReceiveMsgTimeout(MU_Type *base, uint32_t regIndex, uint32_t *readValue)

Blocks to receive a message with timeout protection.

This function waits until the RX register is full and receives the message. If MU1_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

This function provides the same blocking behavior as MU_ReceiveMsg() but with additional timeout protection to prevent system hangs if the other core becomes unresponsive or if hardware issues occur.

Note: Both MU_ReceiveMsg() and MU_ReceiveMsgTimeout() are blocking functions. The difference is that this function includes timeout protection while MU_ReceiveMsg() waits indefinitely.

Parameters

- base – MU peripheral base address.
- regIndex – RX register index, see mu_msg_reg_index_t.
- readValue – Pointer to store the received message.

Return values

- kStatus_Success – Message received successfully.
- kStatus_InvalidArgument – Invalid readValue pointer.
- kStatus_Timeout – Timeout occurred while waiting for RX register to be full.

Returns

status_t

uint32_t MU_ReceiveMsg(MU_Type *base, uint32_t regIndex)

Blocks to receive a message (infinite wait, no timeout protection).

This function waits until the RX register is full and receives the message. This function will wait indefinitely until a message is received.

Note: Both `MU_ReceiveMsg()` and `MU_ReceiveMsgTimeout()` are blocking functions. The difference is that `MU_ReceiveMsgTimeout()` includes timeout protection while this function waits indefinitely.

Warning: This function does not include timeout protection and may cause system hangs if the other core becomes unresponsive. For applications requiring timeout protection, use `MU_ReceiveMsgTimeout()` instead.

Parameters

- `base` – MU peripheral base address.
- `regIndex` – RX register index, see `mu_msg_reg_index_t`.

Returns

The received message.

static inline void `MU_SetFlagsNonBlocking(MU_Type *base, uint32_t flags)`

Sets the 3-bit MU flags reflect on the other MU side.

This function sets the 3-bit MU flags directly. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. The upper layer should make sure the status flag `kMU_FlagsUpdatingFlag` is cleared before calling this function.

```
while (kMU_FlagsUpdatingFlag & MU_GetStatusFlags(base))
{
    } Wait for previous MU flags updating.
MU_SetFlagsNonBlocking(base, 0U); Set the mU flags.
```

Parameters

- `base` – MU peripheral base address.
- `flags` – The 3-bit MU flags to set.

status_t `MU_SetFlags(MU_Type *base, uint32_t flags)`

brief Blocks setting the 3-bit MU flags reflect on the other MU side.

This function blocks setting the 3-bit MU flags. Every time the 3-bit MU flags are changed, the status flag `kMU_FlagsUpdatingFlag` asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag `kMU_FlagsUpdatingFlag` is cleared by hardware. During the flags updating period, the flags cannot be changed. This function waits for the MU status flag `kMU_FlagsUpdatingFlag` cleared and sets the 3-bit MU flags.

If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout`.

return status_t retval `kStatus_Success` Flags were set successfully. `retval kStatus_Timeout` Timeout occurred while waiting for flags to update.

Parameters

- `base` – MU peripheral base address.
- `flags` – The 3-bit MU flags to set.

static inline uint32_t MU_GetFlags(MU_Type *base)

Gets the current value of the 3-bit MU flags set by the other side.

This function gets the current 3-bit MU flags on the current side.

Parameters

- base – MU peripheral base address.

Returns

flags Current value of the 3-bit flags.

static inline uint32_t MU_GetCoreStatusFlags(MU_Type *base)

Gets the MU core status flags.

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU status flags, see `_mu_core_status_flags`.

uint32_t MU_GetStatusFlags(MU_Type *base)

Gets the MU status flags.

This function returns the bit mask of the MU status flags. See `_mu_status_flags`.

```
uint32_t flags;
flags = MU_GetStatusFlags(base); Get all status flags.
if (kMU_Tx0EmptyFlag & flags)
{
    The TX0 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG0_VAL);
}
if (kMU_Tx1EmptyFlag & flags)
{
    The TX1 register is empty. Message can be sent.
    MU_SendMsgNonBlocking(base, kMU_MsgReg1, MSG1_VAL);
}
```

If there are more than 4 general purpose interrupts, use `MU_GetGeneralPurposeStatusFlags`.

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU status flags, see `_mu_status_flags`.

static inline uint32_t MU_GetInterruptsPending(MU_Type *base)

Gets the MU IRQ pending status of enabled interrupts.

This function returns the bit mask of the pending MU IRQs of enabled interrupts. Only these flags are checked. `kMU_Tx0EmptyFlag` `kMU_Tx1EmptyFlag`
`kMU_Tx2EmptyFlag` `kMU_Tx3EmptyFlag` `kMU_Rx0FullFlag` `kMU_Rx1FullFlag`
`kMU_Rx2FullFlag` `kMU_Rx3FullFlag` `kMU_GenInt0Flag` `kMU_GenInt1Flag`
`kMU_GenInt2Flag` `kMU_GenInt3Flag`

Parameters

- base – MU peripheral base address.

Returns

Bit mask of the MU IRQs pending.

```
static inline void MU_ClearStatusFlags(MU_Type *base, uint32_t flags)
```

Clears the specific MU status flags.

This function clears the specific MU status flags. The flags to clear should be passed in as bit mask. See `_mu_status_flags`.

Clear general interrupt 0 and general interrupt 1 pending flags.
`MU_ClearStatusFlags(base, kMU_GenInt0Flag | kMU_GenInt1Flag);`

If there are more than 4 general purpose interrupts, use `MU_ClearGeneralPurposeStatusFlags`.

Parameters

- base – MU peripheral base address.
- flags – Bit mask of the MU status flags. See `_mu_status_flags`. Only the following flags can be cleared by software (if applicable for particular device), other flags are cleared by hardware:
 - kMU_GenInt0Flag
 - kMU_GenInt1Flag
 - kMU_GenInt2Flag
 - kMU_GenInt3Flag
 - kMU_MuResetInterruptFlag
 - kMU_OtherSideEnterRunInterruptFlag
 - kMU_OtherSideEnterHaltInterruptFlag
 - kMU_OtherSideEnterWaitInterruptFlag
 - kMU_OtherSideEnterStopInterruptFlag
 - kMU_OtherSideEnterPowerDownInterruptFlag
 - kMU_ResetAssertInterruptFlag
 - kMU_HardwareResetInterruptFlag

```
static inline void MU_EnableInterrupts(MU_Type *base, uint32_t interrupts)
```

Enables the specific MU interrupts.

This function enables the specific MU interrupts. The interrupts to enable should be passed in as bit mask. See `_mu_interrupt_enable`.

Enable general interrupt 0 and TX0 empty interrupt.
`MU_EnableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

If there are more than 4 general purpose interrupts, use `MU_EnableGeneralPurposeInterrupts`.

Parameters

- base – MU peripheral base address.
- interrupts – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

```
static inline void MU_DisableInterrupts(MU_Type *base, uint32_t interrupts)
```

Disables the specific MU interrupts.

This function disables the specific MU interrupts. The interrupts to disable should be passed in as bit mask. See `_mu_interrupt_enable`.

Disable general interrupt 0 and TX0 empty interrupt.
`MU_DisableInterrupts(base, kMU_GenInt0InterruptEnable | kMU_Tx0EmptyInterruptEnable);`

If there are more than 4 general purpose interrupts, use `MU_DisableGeneralPurposeInterrupts`.

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the MU interrupts. See `_mu_interrupt_enable`.

`status_t` `MU_TriggerInterrupts(MU_Type *base, uint32_t interrupts)`

Triggers interrupts to the other core.

This function triggers the specific interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `_mu_interrupt_trigger`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
if (kStatus_Success != MU_TriggerInterrupts(base, kMU_GenInt0InterruptTrigger | kMU_
↪GenInt2InterruptTrigger))
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

If there are more than 4 general purpose interrupts, use `MU_TriggerGeneralPurposeInterrupts`.

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the interrupts to trigger. See `_mu_interrupt_trigger`.

Return values

- `kStatus_Success` – Interrupts have been triggered successfully.
- `kStatus_Fail` – Previous interrupts have not been accepted.

`status_t` `MU_TriggerNmi(MU_Type *base)`

Triggers NMI to the other core.

This function triggers the NMI to the other core. The MU should not trigger NMI 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.

Parameters

- `base` – MU peripheral base address.

Return values

- `kStatus_Success` – Interrupts have been triggered successfully.
- `kStatus_Fail` – Previous interrupts have not been accepted.

`static inline void` `MU_ClearNmi(MU_Type *base)`

Clear non-maskable interrupt (NMI) sent by the other core.

This function clears non-maskable interrupt (NMI) sent by the other core.

Parameters

- `base` – MU peripheral base address.

```
static inline void MU_EnableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)
```

Enables the MU general purpose interrupts.

This function enables the MU general purpose interrupts. The interrupts to enable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_EnableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to enable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↳GeneralPurposeInterrupt3);
```

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the MU general purpose interrupts, see `mu_general_purpose_interrupt_t`.

```
static inline void MU_DisableGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)
```

Disables the MU general purpose interrupts.

This function disables the MU general purpose interrupts. The interrupts to disable should be passed in as bit mask of `mu_general_purpose_interrupt_t`. The function `MU_DisableInterrupts` only support general interrupt 0~3, this function supports all general interrupts.

For example, to disable general purpose interrupt 0 and 3, use like this:

```
MU_EnableGeneralPurposeInterrupts(MU, kMU_GeneralPurposeInterrupt0 | kMU_
↳GeneralPurposeInterrupt3);
```

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the MU general purpose interrupts. see `mu_general_purpose_interrupt_t`.

```
static inline uint32_t MU_GetGeneralPurposeStatusFlags(MU_Type *base)
```

Gets the MU general purpose interrupt status flags.

This function returns the bit mask of the MU general purpose interrupt status flags. `MU_GetStatusFlags` can only get general purpose interrupt status 0~3, this function can get all general purpose interrupts status.

This example shows to check whether general purpose interrupt 0 and 3 happened.

```
uint32_t flags;
flags = MU_GetGeneralPurposeStatusFlags(base);
if (kMU_GeneralPurposeInterrupt0 & flags)
{
}
if (kMU_GeneralPurposeInterrupt3 & flags)
{
}
```

Parameters

- `base` – MU peripheral base address.

Returns

Bit mask of the MU general purpose interrupt status flags.

```
static inline void MU_ClearGeneralPurposeStatusFlags(MU_Type *base, uint32_t flags)
```

Clear the MU general purpose interrupt status flags.

This function clears the specific MU general purpose interrupt status flags. The flags to clear should be passed in as bit mask. `mu_general_purpose_interrupt_t_mu_status_flags`.

Example to clear general purpose interrupt 0 and general interrupt 1 pending flags.

```
MU_ClearGeneralPurposeStatusFlags(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt1);
```

Parameters

- `base` – MU peripheral base address.
- `flags` – Bit mask of the MU general purpose interrupt status flags. See `mu_general_purpose_interrupt_t`.

```
static inline uint32_t MU_GetRxStatusFlags(MU_Type *base)
```

Return the RX status flags in reverse numerical order.

This function return the RX status flags in reverse order. Note: RFn bits of SR[3-0](mu status register) are mapped in ascending numerical order: RF0 -> SR[0] RF1 -> SR[1] RF2 -> SR[2] RF3 -> SR[3] This function will return these bits in reverse numerical order(RF3->RF1) to comply with `MU_GetRxStatusFlags()` of mu driver. See `MU_GetRxStatusFlags()` from `drivers/mu/fsl_mu.h`

```
status_reg = MU_GetRxStatusFlags(base);
```

Parameters

- `base` – MU peripheral base address.

Returns

MU RX status flags in reverse order

```
status_t MU_TriggerGeneralPurposeInterrupts(MU_Type *base, uint32_t interrupts)
```

Triggers general purpose interrupts to the other core.

This function triggers the specific general purpose interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `mu_general_purpose_interrupt_t`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
status_t status;
status = MU_TriggerGeneralPurposeInterrupts(base, kMU_GeneralPurposeInterrupt0 | kMU_
↪GeneralPurposeInterrupt2);

if (kStatus_Success != status)
{
    Previous general purpose interrupt 0 or general purpose interrupt 2
    has not been processed by the other core.
}
```

Parameters

- `base` – MU peripheral base address.
- `interrupts` – Bit mask of the interrupts to trigger. See `mu_general_purpose_interrupt_t`.

Return values

- `kStatus_Success` – Interrupts have been triggered successfully.

- kStatus_Fail – Previous interrupts have not been accepted.

void MU_BootOtherCore(MU_Type *base, mu_core_boot_mode_t mode)

Boots the other core.

This function boots the other core with a boot configuration.

Parameters

- base – MU peripheral base address.
- mode – The other core boot mode.

void MU_HoldOtherCoreReset(MU_Type *base)

Holds the other core reset.

This function causes the other core to be held in reset following any reset event.

Parameters

- base – MU peripheral base address.

static inline status_t MU_ResetBothSides(MU_Type *base)

Resets the MU for both A side and B side.

This function resets the MU for both A side and B side. Before reset, it is recommended to interrupt processor B, because this function may affect the ongoing processor B programs.

If MU1_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations if waiting for the other side to come out of reset takes too long.

Note: For some platforms, only MU side A could use this function, check reference manual for details.

Parameters

- base – MU peripheral base address.

Return values

- kStatus_Success – The MU was reset successfully.
- kStatus_Timeout – Timeout occurred while waiting for the other side to come out of reset.

Returns

status_t

static inline void MU_SetClockOnOtherCoreEnable(MU_Type *base, bool enable)

Enables or disables the clock on the other core.

This function enables or disables the platform clock on the other core when that core enters a stop mode. If disabled, the platform clock for the other core is disabled when it enters stop mode. If enabled, the platform clock keeps running on the other core in stop mode, until this core also enters stop mode.

Parameters

- base – MU peripheral base address.
- enable – Enable or disable the clock on the other core.

status_t MU_HardwareResetOtherCore(MU_Type *base, bool waitReset, bool holdReset, mu_core_boot_mode_t bootMode)

Hardware reset the other core.

This function resets the other core, the other core could mask the hardware reset by calling `MU_MaskHardwareReset`. The hardware reset mask feature is only available for some platforms. This function could be used together with `MU_BootOtherCore` to control the other core reset workflow.

If `MU1_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout` if waiting for the other core to enter or exit reset takes too long.

Example 1: Reset the other core, and no hold reset

```
MU_HardwareResetOtherCore(MU_A, true, false, bootMode);
```

In this example, the core at MU side B will reset with the specified boot mode.

Example 2: Reset the other core and hold it, then boot the other core later. Here the other core enters reset, and the reset is hold

```
MU_HardwareResetOtherCore(MU_A, true, true, modeDontCare);
```

Current core boot the other core when necessary.

```
MU_BootOtherCore(MU_A, bootMode);
```

Note: The feature `waitReset`, `holdReset`, and `bootMode` might be not supported for some platforms. `waitReset` is only available for platforms that `FSL_FEATURE_MU_NO_CORE_STATUS` not defined as 1 and `FSL_FEATURE_MU_HAS_RESET_ASSERT_INT` not defined as 0. `holdReset` is only available for platforms that `FSL_FEATURE_MU_HAS_RSTH` not defined as 0. `bootMode` is only available for platforms that `FSL_FEATURE_MU_HAS_BOOT` not defined as 0.

Parameters

- `base` – MU peripheral base address.
- `waitReset` – Wait the other core enters reset. Only work when there is `CSSR0[RAIP]`.
 - `true`: Wait until the other core enters reset, if the other core has masked the hardware reset, then this function will be blocked.
 - `false`: Don't wait the reset.
- `holdReset` – Hold the other core reset or not. Only work when there is `CCR0[RSTH]`.
 - `true`: Hold the other core in reset, this function returns directly when the other core enters reset.
 - `false`: Don't hold the other core in reset, this function waits until the other core out of reset.
- `bootMode` – Boot mode of the other core, if `holdReset` is true, this parameter is useless.

Return values

- `kStatus_Success` – The other core was reset successfully.
- `kStatus_Timeout` – Timeout occurred while waiting for the other core to enter or exit reset.

Returns

status_t

FSL_MU_DRIVER_VERSION

MU driver version.

enum __mu_status_flags

MU status flags.

Values:

enumerator kMU_Tx0EmptyFlag

TX0 empty.

enumerator kMU_Tx1EmptyFlag

TX1 empty.

enumerator kMU_Tx2EmptyFlag

TX2 empty.

enumerator kMU_Tx3EmptyFlag

TX3 empty.

enumerator kMU_Rx0FullFlag

RX0 full.

enumerator kMU_Rx1FullFlag

RX1 full.

enumerator kMU_Rx2FullFlag

RX2 full.

enumerator kMU_Rx3FullFlag

RX3 full.

enumerator kMU_GenInt0Flag

General purpose interrupt 0 pending.

enumerator kMU_GenInt1Flag

General purpose interrupt 1 pending.

enumerator kMU_GenInt2Flag

General purpose interrupt 2 pending.

enumerator kMU_GenInt3Flag

General purpose interrupt 3 pending.

enumerator kMU_CoreEventPendingFlag

The other core mode entry event 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.

enumerator kMU__OtherSideEnterRunInterruptFlag

The other side enters run mode.

enumerator kMU__OtherSideEnterHaltInterruptFlag

The other side enters halt mode.

enumerator kMU__OtherSideEnterWaitInterruptFlag

The other side enters wait mode.

enumerator kMU__OtherSideEnterStopInterruptFlag

The other side enters stop mode.

enumerator kMU__OtherSideEnterPowerDownInterruptFlag

The other side enters power down mode.

enumerator kMU__ResetAssertInterruptFlag

The other core reset assert interrupt.

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__OtherSideEnterRunInterruptEnable

The other side enters run mode.

enumerator kMU__OtherSideEnterHaltInterruptEnable

The other side enters halt mode.

enumerator kMU__OtherSideEnterWaitInterruptEnable

The other side enters wait mode.

enumerator kMU__OtherSideEnterStopInterruptEnable

The other side enters stop mode.

enumerator kMU__OtherSideEnterPowerDownInterruptEnable

The other side enters power down mode.

enumerator kMU__ResetAssertInterruptEnable

The other core reset assert interrupt.

enumerator kMU__HardwareResetInterruptEnable

Current side has been hardware reset by the other side.

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_core_status_flags

MU core status flags.

Values:

enumerator kMU__OtherSideEnterRunFlag

The other side in run mode.

enumerator kMU__OtherSideEnterHaltFlag

The other side in halt mode.

enumerator kMU__OtherSideEnterWaitFlag

The other side in wait mode.

enumerator kMU__OtherSideEnterStopFlag

The other side in stop mode.

enumerator kMU_OtherSideEnterPowerDownFlag

The other side in power down mode.

enumerator kMU_OtherSideEnterResetFlag

The other core entered reset.

enumerator kMU_HardwareResetFlag

Current side has been hardware reset by the other side.

enum _mu_msg_reg_index

MU message register index.

Values:

enumerator kMU_MsgReg0

Message register 0.

enumerator kMU_MsgReg1

Message register 1.

enumerator kMU_MsgReg2

Message register 2.

enumerator kMU_MsgReg3

Message register 3.

enum _mu_general_purpose_interrupt

MU general purpose interrupts.

Values:

enumerator kMU_GeneralPurposeInterrupt0

General purpose interrupt 0

enumerator kMU_GeneralPurposeInterrupt1

General purpose interrupt 1

enumerator kMU_GeneralPurposeInterrupt2

General purpose interrupt 2

enumerator kMU_GeneralPurposeInterrupt3

General purpose interrupt 3

typedef enum _mu_msg_reg_index mu_msg_reg_index_t

MU message register index.

typedef enum _mu_general_purpose_interrupt mu_general_purpose_interrupt_t

MU general purpose interrupts.

MU_CORE_INTR(intr)

MU_MISC_INTR(intr)

MU_TX_INTR(intr)

MU_RX_INTR(intr)

MU_GI_INTR(intr)

MU_GET_CORE_INTR(intrs)

MU_GET_TX_INTR(intrs)

MU_GET_RX_INTR(intrs)

MU_GET_GI_INTR(*intrs*)

MU_CORE_FLAG(*flag*)

MU_STAT_FLAG(*flag*)

MU_TX_FLAG(*flag*)

MU_RX_FLAG(*flag*)

MU_GI_FLAG(*flag*)

MU_GET_CORE_FLAG(*flags*)

MU_GET_STAT_FLAG(*flags*)

MU_GET_TX_FLAG(*flags*)

MU_GET_RX_FLAG(*flags*)

MU_GET_GI_FLAG(*flags*)

FSL_FEATURE_MU_GPI_COUNT

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.

2.49 OTFAD: On The Fly AES-128 Decryption Driver

void OTFAD_GetDefaultConfig(*otfad_config_t* **config*)

OTFAD module initialization function.

Parameters

- *config* – OTFAD configuration.

void OTFAD_Init(OTFAD_Type **base*, const *otfad_config_t* **config*)

OTFAD module initialization function.

Parameters

- *base* – OTFAD base address.
- *config* – OTFAD configuration.

void OTFAD_Deinit(OTFAD_Type **base*)

Deinitializes the OTFAD.

static inline uint32_t OTFAD_GetOperateMode(OTFAD_Type **base*)

OTFAD module get operate mode.

Parameters

- *base* – OTFAD base address.

static inline uint32_t OTFAD_GetStatus(OTFAD_Type *base)

OTFAD module get status.

Parameters

- base – OTFAD base address.

status_t OTFAD_SetEncryptionConfig(OTFAD_Type *base, const *otfad_encryption_config_t* *config)

OTFAD module set encryption configuration.

Note: if enable keyblob process, the first 256 bytes external memory is use for keyblob data, so this region shouldn't be in OTFAD region.

Parameters

- base – OTFAD base address.
- config – encryption configuration.

status_t OTFAD_GetEncryptionConfig(OTFAD_Type *base, *otfad_encryption_config_t* *config)

OTFAD module get encryption configuration.

Note: if enable keyblob process, the first 256 bytes external memory is use for keyblob data, so this region shouldn't be in OTFAD region.

Parameters

- base – OTFAD base address.
- config – encryption configuration.

status_t OTFAD_HitDetermination(OTFAD_Type *base, uint32_t address, uint8_t *contextIndex)

OTFAD module do hit determination.

Parameters

- base – OTFAD base address.
- address – the physical address space assigned to the QuadSPI(FlexSPI) module.
- contextIndex – hitted context region index.

Returns

status, such as kStatus_Success or kStatus_OTFAD_ResRegAccessMode.

FSL_OTFAD_DRIVER_VERSION

Driver version.

Status codes for the OTFAD driver.

Values:

enumerator kStatus_OTFAD_ResRegAccessMode

Restricted register mode

enumerator kStatus_OTFAD_AddressError

End address less than start address

enumerator kStatus_OTFAD_RegionOverlap

the OTFAD does not support any form of memory region overlap, for system accesses that hit in multiple contexts or no contexts, the fetched data is simply bypassed

enumerator kStatus_OTFAD_RegionMiss

For accesses that hit in a single context, but not the selected one

OTFAD context type.

Values:

enumerator kOTFAD_Context_0
context 0

enumerator kOTFAD_Context_1
context 1

enumerator kOTFAD_Context_2
context 2

enumerator kOTFAD_Context_3
context 3

OTFAD operate mode.

Values:

enumerator kOTFAD_NRM
Normal Mode

enumerator kOTFAD_SVM
Security Violation Mode

enumerator kOTFAD_LDM
Logically Disabled Mode

typedef struct *_otfad_encryption_config* otfad_encryption_config_t
OTFAD encryption configuration structure.

typedef struct *_otfad_config* otfad_config_t
OTFAD configuration structure.

struct *_otfad_encryption_config*
#include <fsl_otfad.h> OTFAD encryption configuration structure.

Public Members

bool valid

The context is valid or not

bool AESdecryption

AES decryption enable

uint8_t readOnly

read write attribute for the entire set of context registers

uint8_t contextIndex

OTFAD context index

uint32_t startAddr

Start address

uint32_t endAddr

End address

```
uint32_t key[4]
    Encryption key
uint32_t counter[2]
    Encryption counter
struct __otfad_config
    #include <fsl_otfad.h> OTFAD configuration structure.
```

Public Members

```
bool forceSVM
    Force entry into SVM after a write
bool forceLDM
    Force entry into LDM after a write
bool restrictedRegAccess
    Restricted register access enable
bool enableOTFAD
    OTFAD has decryption enabled
```

2.50 PDM: Microphone Interface

2.51 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_EnableDoze(PDM_Type *base, bool enable)

Enables/disables DOZE.

Parameters

- base – PDM base pointer
- enable – True means the module will enter Disable/Low Leakage mode when ipg_doze is asserted, false means the module will not enter Disable/Low Leakage mode when ipg_doze is asserted.

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_GetOutputStatus(PDM_Type *base)
```

Gets the PDM output status flag. Use the Status Mask in `_pdm_output_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_ClearOutputStatus(PDM_Type *base, uint32_t mask)
```

Clears the PDM output status.

Parameters

- base – PDM base pointer
- mask – State mask. It can be a combination of the status in `_pdm_output_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 uint32_t PDM_GetHwvadVoiceDetectedFlag(PDM_Type *base)

Get the PDM voice activity detector voice detected flags. NOTE: this flag is auto cleared when voice gone.

Parameters

- base – PDM base pointer

Returns

voice detected 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_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_StatusFIRFilterReady

FIR filter data is ready

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

enumerator kPDM_StatusCh4FifoDataAvaliable

channel 4 fifo data reached watermark level

enumerator kPDM_StatusCh5FifoDataAvaliable

channel 5 fifo data reached watermark level

enumerator kPDM_StatusCh6FifoDataAvaliable
channel 6 fifo data reached watermark level

enumerator kPDM_StatusCh7FifoDataAvaliable
channel 7 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_EnableChannel4
channgel 4 enable mask

enumerator kPDM_EnableChannel5
channgel 5 enable mask

enumerator kPDM_EnableChannel6
channgel 6 enable mask

enumerator kPDM_EnableChannel7
channgel 7 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_FifoStatusUnderflowCh4
channel4 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh5
channel5 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh6
channel6 fifo status underflow

enumerator kPDM_FifoStatusUnderflowCh7
channel7 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

enumerator kPDM_FifoStatusOverflowCh4
channel4 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh5
channel5 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh6
channel6 fifo status overflow

enumerator kPDM_FifoStatusOverflowCh7
channel7 fifo status overflow

enum _pdm_output_status

The PDM output status.

Values:

enumerator kPDM_OutputStatusUnderFlowCh0
channel0 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh1
channel1 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh2
channel2 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh3
channel3 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh4
channel4 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh5
channel5 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh6
channel6 output status underflow

enumerator kPDM_OutputStatusUnderFlowCh7
channel7 output status underflow

enumerator kPDM_OutputStatusOverFlowCh0
channel0 output status overflow

enumerator kPDM_OutputStatusOverFlowCh1
channel1 output status overflow

enumerator kPDM_OutputStatusOverflowCh2
channel2 output status overflow

enumerator kPDM_OutputStatusOverflowCh3
channel3 output status overflow

enumerator kPDM_OutputStatusOverflowCh4
channel4 output status overflow

enumerator kPDM_OutputStatusOverflowCh5
channel5 output status overflow

enumerator kPDM_OutputStatusOverflowCh6
channel6 output status overflow

enumerator kPDM_OutputStatusOverflowCh7
channel7 output status overflow

enum __pdm_dc_removal
PDM DC removal configurations.

Values:

enumerator kPDM_DcRemovalCutOff21Hz
DC removal cut off 21HZ

enumerator kPDM_DcRemovalCutOff83Hz
DC removal cut off 83HZ

enumerator kPDM_DcRemovalCutOff152Hz
DC removal cut off 152HZ

enumerator kPDM_DcRemovalBypass
DC removal bypass

enum __pdm_df_quality_mode
PDM decimation filter quality mode.

Values:

enumerator kPDM_QualityModeMedium
quality mode medium

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_quality_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_DataWidth16
PDM data width 16bit

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 removal 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 cutOffFreq
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

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

2.52 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.

- Scatter gather case: This function support dynamic scatter gather and static 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],
```

(continues on next page)

(continued from previous page)

```
.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

2.53 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__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 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_SetAlphaSurfaceOverlayColorKey(PXP_Type *base, 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 p colorKeyLow to p 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.
- colorKeyLow – Color key low range.
- colorKeyHigh – Color key high range.

static inline void PXP_EnableAlphaSurfaceOverlayColorKey(PXP_Type *base, bool enable)

Enable or disable the alpha surface color key.

Parameters

- base – PXP peripheral base address.
- 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, uint32_t backGroundColor)

Set the back ground color of PS.

Parameters

- base – PXP peripheral base address.
- 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, 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 colorKeyHigh to 0 will disable the colorkeying.

Parameters

- base – PXP peripheral base address.
- 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 outpt 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:

- a. Prepare the PXP register values except STAT, CSCCOEFn, NEXT in the memory in the order they appear in the register map.
- b. Call this function sets the new operation to PXP.
- c. 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_SetCsc2Config(PXP_Type *base, const *pxp_csc2_config_t* *config)

Set the CSC2 configuration.

The CSC2 module receives pixels in any color space and can convert the pixels into any of RGB, YUV, or YCbCr color spaces. The output pixels are passed onto the LUT and rotation engine for further processing

Parameters

- base – PXP peripheral base address.
- config – Pointer to the configuration.

static inline void PXP_EnableCsc2(PXP_Type *base, bool enable)

Enable or disable the CSC2.

Parameters

- base – PXP peripheral base address.
- enable – True to enable, false to disable.

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_SetLutConfig(PXP_Type *base, const *pxp_lut_config_t* *config)

Set the LUT configuration.

The lookup table (LUT) is used to modify pixels in a manner that is not linear and that cannot be achieved by the color space conversion modules. To setup the LUT, the complete workflow is:

- a. Use PXP_SetLutConfig to set the configuration, such as the lookup mode.
- b. Use PXP_LoadLutTable to load the lookup table to PXP.
- c. Use PXP_EnableLut to enable the function.

Parameters

- base – PXP peripheral base address.
- config – Pointer to the configuration.

status_t PXP_LoadLutTable(PXP_Type *base, *pxp_lut_lookup_mode_t* lookupMode, uint32_t bytesNum, uint32_t memAddr, uint16_t lutStartAddr)

Set the look up table to PXP.

If lookup mode is DIRECT mode, this function loads bytesNum of values from the address memAddr into PXP LUT address lutStartAddr. So this function allows only update part of the PXP LUT.

If lookup mode is CACHE mode, this function sets the new address to `memAddr` and invalid the PXP LUT cache.

Parameters

- `base` – PXP peripheral base address.
- `lookupMode` – Which lookup mode is used. Note that this parameter is only used to distinguish DIRECT mode and CACHE mode, it does not change the register value `PXP_LUT_CTRL[LOOKUP_MODE]`. To change that value, use function `PXP_SetLutConfig`.
- `bytesNum` – How many bytes to set. This value must be divisible by 8.
- `memAddr` – Address of look up table to set.
- `lutStartAddr` – The LUT value will be loaded to LUT from index `lutAddr`. It should be 8 bytes aligned.

Return values

- `kStatus__Success` – Load successfully.
- `kStatus__InvalidArgument` – Failed because of invalid argument.

```
static inline void PXP_EnableLut(PXP_Type *base, bool enable)
```

Enable or disable the LUT.

Parameters

- `base` – PXP peripheral base address.
- `enable` – True to enable, false to disable.

```
static inline void PXP_Select8kLutBank(PXP_Type *base, pxp_lut_8k_bank_t bank)
```

Select the 8kB LUT bank in DIRECT_RGB444 mode.

Parameters

- `base` – PXP peripheral base address.
- `bank` – The bank to select.

```
void PXP_SetPorterDuffConfig(PXP_Type *base, const pxp_porter_duff_config_t *config)
```

Set the Porter Duff configuration.

Parameters

- `base` – PXP peripheral base address.
- `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 extended `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_LutDmaLoadInterruptEnable`

The LUT table has been loaded by DMA. bit 3

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_LutDmaLoadFlag
The LUT table has been loaded by DMA. bit 8

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__PsPixelFormatARGB8888
        32-bit pixels with alpha(when participates in blend with
        alpha surface uses pixel format that has alpha value) or without alpha (unpacked 24-
        bit format)
    enumerator kPXP__PsPixelFormatARGB1555
        16-bit pixels with alpha(when participates in blend with
        alpha surface uses pixel format that has alpha value) or without alpha.
    enumerator kPXP__PsPixelFormatARGB4444
        16-bit pixels with alpha(when participates in blend with
        alpha surface uses pixel format that has alpha value) or 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)

enumerator kPXP_PsPixelFormatRGBA8888
32-bit pixels with alpha at low 8-bit

enumerator kPXP_PsPixelFormatRGBA5551
16-bit pixels with alpha at low 1-bit.

enumerator kPXP_PsPixelFormatRGBA4444
16-bit pixels with alpha at low 4-bit.

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.

enumerator kPXP_AsPixelFormatRGBA8888
32-bit pixels with alpha at low 8-bit.

enumerator kPXP_AsPixelFormatRGBA5551
16-bit pixels with alpha at low 1-bit.

enumerator kPXP_AsPixelFormatRGBA4444
16-bit pixels with alpha at low 4-bit.

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_lut_lookup_mode
PXP LUT lookup mode.

Values:

enumerator kPXP_LutCacheRGB565
LUT ADDR = R[7:3],G[7:2],B[7:3]. Use all 16KB of LUT for indirect cached 128KB lookup.

enumerator kPXP_LutDirectY8
LUT ADDR = 16'b0,Y[7:0]. Use the first 256 bytes of LUT. Only third data path byte is transformed.

enumerator kPXP_LutDirectRGB444
LUT ADDR = R[7:4],G[7:4],B[7:4]. Use one 8KB bank of LUT selected by PXP_Select8kLutBank.

enumerator kPXP_LutDirectRGB454
LUT ADDR = R[7:4],G[7:3],B[7:4]. Use all 16KB of LUT.

enum _pxp_lut_out_mode

PXP LUT output mode.

Values:

enumerator kPXP_LutOutY8

R/Y byte lane 2 lookup, bytes 1,0 bypassed.

enumerator kPXP_LutOutRGBW4444CFA

Byte lane 2 = CFA_Y8, byte lane 1,0 = RGBW4444.

enumerator kPXP_LutOutRGB888

RGB565->RGB888 conversion for Gamma correction.

enum _pxp_lut_8k_bank

PXP LUT 8K bank index used when lookup mode is kPXP_LutDirectRGB444.

Values:

enumerator kPXP_Lut8kBank0

The first 8K bank used.

enumerator kPXP_Lut8kBank1

The second 8K bank used.

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.

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

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 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
```

XPX 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_lut_lookup_mode pxp_lut_lookup_mode_t
```

XPX LUT lookup mode.

```
typedef enum _pxp_lut_out_mode pxp_lut_out_mode_t
```

XPX LUT output mode.

```
typedef enum _pxp_lut_8k_bank pxp_lut_8k_bank_t
```

XPX LUT 8K bank index used when lookup mode is kXPX_LutDirectRGB444.

```
typedef struct _pxp_lut_config pxp_lut_config_t
```

XPX LUT configuration.

```
typedef enum _pxp_ram pxp_ram_t
```

XPX internal memory.

```
typedef struct _pxp_dither_final_lut_data pxp_dither_final_lut_data_t
```

XPX dither final LUT data.

```
typedef struct _pxp_dither_config pxp_dither_config_t
```

XPX dither configuration.

```
typedef enum _pxp_porter_duff_blend_mode pxp_porter_duff_blend_mode_t
```

XPX Porter Duff blend mode. Note: don't change the enum item value.

```
typedef struct _pxp_pic_copy_config pxp_pic_copy_config_t
```

XPX Porter Duff blend mode. Note: don't change the enum item value.

XPX_LUT_TABLE_BYTE

XPX_INTERNAL_RAM_LUT_BYTE

XPX_SHARE_ROTATE

XPX_USE_PATH

XPX_COMBINE_BYTE_TO_WORD(dataAddr)

```
struct _pxp_output_buffer_config
```

#include <fsl_pxp.h> XPX 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_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_lut_config

#include <fsl_pxp.h> PXP LUT configuration.

Public Members

pxp_lut_lookup_mode_t lookupMode

Look up mode.

pxp_lut_out_mode_t outMode

Out mode.

uint32_t cfaValue

The CFA value used when look up mode is kPXP_LutOutRGBW4444CFA.

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.

2.54 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

`typedef enum _rgpio_pin_direction rgpio_pin_direction_t`
RGPIO direction definition.

`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

2.55 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.

2.56 SAI: Serial Audio Interface

2.57 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_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 –

static inline void SAI_TxEnableInterrupts(I2S_Type *base, uint32_t mask)

Enables the SAI Tx interrupt requests.

Parameters

- base – SAI base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSAI_WordStartInterruptEnable
 - kSAI_SyncErrorInterruptEnable
 - kSAI_FIFOWarningInterruptEnable
 - kSAI_FIFORequestInterruptEnable
 - kSAI_FIFOErrorInterruptEnable

static inline void SAI_RxEnableInterrupts(I2S_Type *base, uint32_t mask)

Enables the SAI Rx interrupt requests.

Parameters

- base – SAI base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSAI_WordStartInterruptEnable
 - kSAI_SyncErrorInterruptEnable
 - kSAI_FIFOWarningInterruptEnable
 - kSAI_FIFORequestInterruptEnable
 - kSAI_FIFOErrorInterruptEnable

static inline void SAI_TxDisableInterrupts(I2S_Type *base, uint32_t mask)

Disables the SAI Tx interrupt requests.

Parameters

- base – SAI base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSAI_WordStartInterruptEnable
 - kSAI_SyncErrorInterruptEnable
 - kSAI_FIFOWarningInterruptEnable

- kSAI_FIFORequestInterruptEnable
- kSAI_FIFOErrorInterruptEnable

static inline void SAI_RxDisableInterrupts(I2S_Type *base, uint32_t mask)

Disables the SAI Rx interrupt requests.

Parameters

- base – SAI base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSAI_WordStartInterruptEnable
 - kSAI_SyncErrorInterruptEnable
 - kSAI_FIFOWarningInterruptEnable
 - kSAI_FIFORequestInterruptEnable
 - kSAI_FIFOErrorInterruptEnable

static inline void SAI_TxEnableDMA(I2S_Type *base, uint32_t mask, bool enable)

Enables/disables the SAI Tx DMA requests.

Parameters

- base – SAI base pointer
- mask – DMA source The parameter can be combination of the following sources if defined.
 - kSAI_FIFOWarningDMAEnable
 - kSAI_FIFORequestDMAEnable
- enable – True means enable DMA, false means disable DMA.

static inline void SAI_RxEnableDMA(I2S_Type *base, uint32_t mask, bool enable)

Enables/disables the SAI Rx DMA requests.

Parameters

- base – SAI base pointer
- mask – DMA source The parameter can be a combination of the following sources if defined.
 - kSAI_FIFOWarningDMAEnable
 - kSAI_FIFORequestDMAEnable
- enable – True means enable DMA, false means disable DMA.

static inline uintptr_t SAI_TxGetDataRegisterAddress(I2S_Type *base, uint32_t channel)

Gets the SAI Tx data register address.

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

- base – SAI base pointer.
- channel – Which data channel used.

Returns

data register address.

```
static inline uintptr_t SAI_RxGetDataRegisterAddress(I2S_Type *base, uint32_t channel)
```

Gets the SAI Rx data register address.

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

- base – SAI base pointer.
- channel – Which data channel used.

Returns

data register address.

```
void SAI_WriteBlocking(I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer,  
                      uint32_t size)
```

Sends data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

```
void SAI_WriteMultiChannelBlocking(I2S_Type *base, uint32_t channel, uint32_t channelMask,  
                                   uint32_t bitWidth, uint8_t *buffer, uint32_t size)
```

Sends data to multi channel using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- channelMask – channel mask.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

```
static inline void SAI_WriteData(I2S_Type *base, uint32_t channel, uint32_t data)
```

Writes data into SAI FIFO.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- data – Data needs to be written.

```
void SAI_ReadBlocking(I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer,  
                    uint32_t size)
```

Receives data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be read.
- size – Bytes to be read.

```
void SAI_ReadMultiChannelBlocking(I2S_Type *base, uint32_t channel, uint32_t channelMask,  
                                uint32_t bitWidth, uint8_t *buffer, uint32_t size)
```

Receives multi channel data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SAI base pointer.
- channel – Data channel used.
- channelMask – channel mask.
- bitWidth – How many bits in an audio word; usually 8/16/24/32 bits.
- buffer – Pointer to the data to be read.
- size – Bytes to be read.

```
static inline uint32_t SAI_ReadData(I2S_Type *base, uint32_t channel)
```

Reads data from the SAI FIFO.

Parameters

- base – SAI base pointer.
- channel – Data channel used.

Returns

Data in SAI FIFO.

```
void SAI_TransferTxCreateHandle(I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t  
                              callback, void *userData)
```

Initializes the SAI Tx handle.

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SAI base pointer
- handle – SAI handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function

```
void SAI_TransferRxCreateHandle(I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t  
                                callback, void *userData)
```

Initializes the SAI Rx handle.

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function.

```
void SAI_TransferTxSetConfig(I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)
```

SAI transmitter transfer configurations.

This function initializes the Tx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.
- config – transmitter configurations.

```
void SAI_TransferRxSetConfig(I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)
```

SAI receiver transfer configurations.

This function initializes the Rx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

- base – SAI base pointer.
- handle – SAI handle pointer.
- config – receiver configurations.

```
status_t SAI_TransferSendNonBlocking(I2S_Type *base, sai_handle_t *handle, sai_transfer_t  
                                      *xfer)
```

Performs an interrupt non-blocking send transfer on SAI.

Note: This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

- base – SAI base pointer.
- handle – Pointer to the sai_handle_t structure which stores the transfer state.
- xfer – Pointer to the sai_transfer_t structure.

Return values

- kStatus_Success – Successfully started the data receive.
- kStatus_SAI_TxBusy – Previous receive still not finished.

- `kStatus_InvalidArgument` – The input parameter is invalid.

status_t SAI_TransferReceiveNonBlocking(I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)

Performs an interrupt non-blocking receive transfer on SAI.

Note: This API returns immediately after the transfer initiates. Call the `SAI_RxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SAI_Busy`, the transfer is finished.

Parameters

- `base` – SAI base pointer
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.
- `xfer` – Pointer to the `sai_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully started the data receive.
- `kStatus_SAI_RxBusy` – Previous receive still not finished.
- `kStatus_InvalidArgument` – The input parameter is invalid.

status_t SAI_TransferGetSendCount(I2S_Type *base, sai_handle_t *handle, size_t *count)

Gets a set byte count.

Parameters

- `base` – SAI base pointer.
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.
- `count` – Bytes count sent.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

status_t SAI_TransferGetReceiveCount(I2S_Type *base, sai_handle_t *handle, size_t *count)

Gets a received byte count.

Parameters

- `base` – SAI base pointer.
- `handle` – Pointer to the `sai_handle_t` structure which stores the transfer state.
- `count` – Bytes count received.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

void SAI_TransferAbortSend(I2S_Type *base, *sai_handle_t* *handle)

Aborts the current send.

Note: This API can be called any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – SAI base pointer.
- handle – Pointer to the *sai_handle_t* structure which stores the transfer state.

void SAI_TransferAbortReceive(I2S_Type *base, *sai_handle_t* *handle)

Aborts the current IRQ receive.

Note: This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – SAI base pointer
- handle – Pointer to the *sai_handle_t* structure which stores the transfer state.

void SAI_TransferTerminateSend(I2S_Type *base, *sai_handle_t* *handle)

Terminate all SAI send.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSend.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferTerminateReceive(I2S_Type *base, *sai_handle_t* *handle)

Terminate all SAI receive.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceive.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferTxHandleIRQ(I2S_Type *base, *sai_handle_t* *handle)

Tx interrupt handler.

Parameters

- base – SAI base pointer.
- handle – Pointer to the *sai_handle_t* structure.

void SAI_TransferRxHandleIRQ(I2S_Type *base, *sai_handle_t* *handle)

Tx interrupt handler.

Parameters

- base – SAI base pointer.

- handle – Pointer to the sai_handle_t structure.

void SAI_DriverIRQHandler(uint32_t instance)

SAI driver IRQ handler common entry.

This function provides the common IRQ request entry for SAI.

Parameters

- instance – SAI instance.

FSL_SAI_DRIVER_VERSION

Version 2.4.10

_sai_status_t, SAI return status.

Values:

enumerator kStatus_SAI_TxBusy
SAI Tx is busy.

enumerator kStatus_SAI_RxBusy
SAI Rx is busy.

enumerator kStatus_SAI_TxError
SAI Tx FIFO error.

enumerator kStatus_SAI_RxError
SAI Rx FIFO error.

enumerator kStatus_SAI_QueueFull
SAI transfer queue is full.

enumerator kStatus_SAI_TxIdle
SAI Tx is idle

enumerator kStatus_SAI_RxIdle
SAI Rx is idle

_sai_channel_mask, sai channel mask value, actual channel numbers is depend soc specific

Values:

enumerator kSAI_Channel0Mask
channel 0 mask value

enumerator kSAI_Channel1Mask
channel 1 mask value

enumerator kSAI_Channel2Mask
channel 2 mask value

enumerator kSAI_Channel3Mask
channel 3 mask value

enumerator kSAI_Channel4Mask
channel 4 mask value

enumerator kSAI_Channel5Mask
channel 5 mask value

enumerator kSAI_Channel6Mask
channel 6 mask value

enumerator kSAI_Channel7Mask
channel 7 mask value

enum _sai_protocol

Define the SAI bus type.

Values:

enumerator kSAI_BusLeftJustified
Uses left justified format.

enumerator kSAI_BusRightJustified
Uses right justified format.

enumerator kSAI_BusI2S
Uses I2S format.

enumerator kSAI_BusPCMA
Uses I2S PCM A format.

enumerator kSAI_BusPCMB
Uses I2S PCM B format.

enum _sai_master_slave

Master or slave mode.

Values:

enumerator kSAI_Master
Master mode include bclk and frame sync

enumerator kSAI_Slave
Slave mode include bclk and frame sync

enumerator kSAI_Bclk_Master_FrameSync_Slave
bclk in master mode, frame sync in slave mode

enumerator kSAI_Bclk_Slave_FrameSync_Master
bclk in slave mode, frame sync in master mode

enum _sai_mono_stereo

Mono or stereo audio format.

Values:

enumerator kSAI_Stereo
Stereo sound.

enumerator kSAI_MonoRight
Only Right channel have sound.

enumerator kSAI_MonoLeft
Only left channel have sound.

enum _sai_data_order

SAI data order, MSB or LSB.

Values:

enumerator kSAI_DataLSB
LSB bit transferred first

enumerator kSAI_DataMSB
MSB bit transferred first

enum `_sai_clock_polarity`

SAI clock polarity, active high or low.

Values:

enumerator `kSAI_PolarityActiveHigh`

Drive outputs on rising edge

enumerator `kSAI_PolarityActiveLow`

Drive outputs on falling edge

enumerator `kSAI_SampleOnFallingEdge`

Sample inputs on falling edge

enumerator `kSAI_SampleOnRisingEdge`

Sample inputs on rising edge

enum `_sai_sync_mode`

Synchronous or asynchronous mode.

Values:

enumerator `kSAI_ModeAsync`

Asynchronous mode

enumerator `kSAI_ModeSync`

Synchronous mode (with receiver or transmit)

enum `_sai_mclk_source`

Master clock source.

Values:

enumerator `kSAI_MclkSourceSysclk`

Master clock from the system clock

enumerator `kSAI_MclkSourceSelect1`

Master clock from source 1

enumerator `kSAI_MclkSourceSelect2`

Master clock from source 2

enumerator `kSAI_MclkSourceSelect3`

Master clock from source 3

enum `_sai_bclk_source`

Bit clock source.

Values:

enumerator `kSAI_BclkSourceBusclk`

Bit clock using bus clock

enumerator `kSAI_BclkSourceMclkOption1`

Bit clock MCLK option 1

enumerator `kSAI_BclkSourceMclkOption2`

Bit clock MCLK option2

enumerator `kSAI_BclkSourceMclkOption3`

Bit clock MCLK option3

enumerator `kSAI_BclkSourceMclkDiv`

Bit clock using master clock divider

enumerator kSAI_BclkSourceOtherSai0

Bit clock from other SAI device

enumerator kSAI_BclkSourceOtherSai1

Bit clock from other SAI device

_sai_interrupt_enable_t, The SAI interrupt enable flag

Values:

enumerator kSAI_WordStartInterruptEnable

Word start flag, means the first word in a frame detected

enumerator kSAI_SyncErrorInterruptEnable

Sync error flag, means the sync error is detected

enumerator kSAI_FIFOWarningInterruptEnable

FIFO warning flag, means the FIFO is empty

enumerator kSAI_FIFOErrorInterruptEnable

FIFO error flag

_sai_dma_enable_t, The DMA request sources

Values:

enumerator kSAI_FIFOWarningDMAEnable

FIFO warning 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_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_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_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_mclk_source sai_mclk_source_t
    Mater clock source.
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 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_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_bit_clock sai_bit_clock_t
    sai bit clock configurations

typedef struct _sai_frame_sync sai_frame_sync_t
    sai frame sync configurations

typedef struct _sai_serial_data sai_serial_data_t
    sai serial data configurations

typedef struct _sai_transceiver sai_transceiver_t
    sai transceiver configurations

typedef struct _sai_transfer sai_transfer_t
    SAI transfer structure.

typedef struct _sai_handle sai_handle_t

typedef void (*sai_transfer_callback_t)(I2S_Type *base, sai_handle_t *handle, status_t status,
void *userData)
    SAI transfer callback prototype.
```

MCUX_SDK_SAI_ALLOW_NULL_FIFO_WATERMARK

Used to control whether SAI_RxSetFifoConfig()/SAI_TxSetFifoConfig() allows a NULL FIFO watermark.

If this macro is set to 0 then SAI_RxSetFifoConfig()/SAI_TxSetFifoConfig() will set the watermark to half of the FIFO's depth if passed a NULL watermark.

MCUX_SDK_SAI_DISABLE_IMPLICIT_CHAN_CONFIG

Disable implicit channel data configuration within SAI_TxSetConfig()/SAI_RxSetConfig().

Use this macro to control whether SAI_RxSetConfig()/SAI_TxSetConfig() will attempt to implicitly configure the channel data. By channel data we mean the startChannel, channelMask, endChannel, and channelNums fields from the sai_transceiver_t structure. By default, SAI_TxSetConfig()/SAI_RxSetConfig() will attempt to compute these fields, which may not be desired in cases where the user wants to set them before the call to said functions.

SAI_XFER_QUEUE_SIZE

SAI transfer queue size, user can refine it according to use case.

FSL_SAI_HAS_FIFO_EXTEND_FEATURE

sai fifo feature

```
struct _sai_config
    #include <fsl_sai.h> SAI user configuration structure.
```

Public Members

```
sai_protocol_t protocol
    Audio bus protocol in SAI

sai_sync_mode_t syncMode
    SAI sync mode, control Tx/Rx clock sync

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

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_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

sai_clock_polarity_t frameSyncPolarity

frame sync polarity


```
struct _sai_serial_data
#include <fsl_sai.h> sai serial data configurations
```

Public Members

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_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

2.58 SAI EDMA Driver

```
void SAI_TransferTxCreateHandleEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                     sai_edma_callback_t callback, void *userData,  
                                     edma_handle_t *txDmaHandle)
```

Initializes the SAI eDMA handle.

This function initializes the SAI master DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- callback – Pointer to user callback function.
- userData – User parameter passed to the callback function.
- txDmaHandle – eDMA handle pointer, this handle shall be static allocated by users.

```
void SAI_TransferRxCreateHandleEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                   sai_edma_callback_t callback, void *userData,  
                                   edma_handle_t *rxDmaHandle)
```

Initializes the SAI Rx eDMA handle.

This function initializes the SAI slave DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- callback – Pointer to user callback function.
- userData – User parameter passed to the callback function.
- rxDmaHandle – eDMA handle pointer, this handle shall be static allocated by users.

```
void SAI_TransferSetInterleaveType(sai_edma_handle_t *handle, sai_edma_interleave_t  
                                  interleaveType)
```

Initializes the SAI interleave type.

This function initializes the SAI DMA handle member interleaveType, it shall be called only when application would like to use type kSAI_EDMAInterleavePerChannelBlock, since the default interleaveType is kSAI_EDMAInterleavePerChannelSample always

Parameters

- handle – SAI eDMA handle pointer.
- interleaveType – Multi channel interleave type.

```
void SAI_TransferTxSetConfigEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                 sai_transceiver_t *saiConfig)
```

Configures the SAI Tx.

Note: SAI eDMA supports data transfer in a multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of sai_transceiver_t with the corresponding values of _sai_channel_mask enum, enable the FIFO Combine mode by assigning kSAI_FifoCombineModeEnabledOnWrite to the fifoCombine member of sai_fifo_combine_t which is a member of sai_transceiver_t. This is an example of multi-channel data transfer configuration step.

```
sai_transceiver_t config;  
SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits, kSAI_Stereo, kSAI_Channel0Mask|kSAI_  
↪Channel1Mask);  
config.fifo.fifoCombine = kSAI_FifoCombineModeEnabledOnWrite;  
SAI_TransferTxSetConfigEDMA(I2S0, &edmaHandle, &config);
```

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- saiConfig – sai configurations.

```
void SAI_TransferRxSetConfigEDMA(I2S_Type *base, sai_edma_handle_t *handle,  
                                sai_transceiver_t *saiConfig)
```

Configures the SAI Rx.

Note: SAI eDMA supports data transfer in a multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of `sai_transceiver_t` with the corresponding values of `_sai_channel_mask` enum, enable the FIFO Combine mode by assigning `kSAI_FifoCombineModeEnabledOnRead` to the `fifoCombine` member of `sai_fifo_combine_t` which is a member of `sai_transceiver_t`. This is an example of multi-channel data transfer configuration step.

```
sai_transceiver_t config;  
SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits, kSAI_Stereo, kSAI_Channel0Mask|kSAI_  
↪Channel1Mask);  
config.fifo.fifoCombine = kSAI_FifoCombineModeEnabledOnRead;  
SAI_TransferRxSetConfigEDMA(I2S0, &edmaHandle, &config);
```

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- saiConfig – sai configurations.

```
status_t SAI_TransferSendEDMA(I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t  
                             *xfer)
```

Performs a non-blocking SAI transfer using DMA.

This function support multi channel transfer,

- a. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
- b. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Note: This interface returns immediately after the transfer initiates. Call `SAI_GetTransferStatus` to poll the transfer status and check whether the SAI transfer is finished.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- xfer – Pointer to the DMA transfer structure.

Return values

- kStatus__Success – Start a SAI eDMA send successfully.
- kStatus__InvalidArgument – The input argument is invalid.
- kStatus__TxBusy – SAI is busy sending data.

status_t SAI_TransferReceiveEDMA(I2S_Type *base, *sai_edma_handle_t* *handle, *sai_transfer_t* *xfer)

Performs a non-blocking SAI receive using eDMA.

This function support multi channel transfer,

- a. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
- b. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Note: This interface returns immediately after the transfer initiates. Call the SAI_GetReceiveRemainingBytes to poll the transfer status and check whether the SAI transfer is finished.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.
- xfer – Pointer to DMA transfer structure.

Return values

- kStatus__Success – Start a SAI eDMA receive successfully.
- kStatus__InvalidArgument – The input argument is invalid.
- kStatus__RxBusy – SAI is busy receiving data.

status_t SAI_TransferSendLoopEDMA(I2S_Type *base, *sai_edma_handle_t* *handle, *sai_transfer_t* *xfer, uint32_t loopTransferCount)

Performs a non-blocking SAI loop transfer using eDMA.

Once the loop transfer start, application can use function SAI_TransferAbortSendEDMA to stop the loop transfer.

Note: This function support loop transfer only,such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in *sai_edma_handle_t*, so application could redefine the SAI_XFER_QUEUE_SIZE to determine the proper TCD pool size. This function support one sai channel only.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- xfer – Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (loopTransferCount).
- loopTransferCount – the counts of xfer array.

Return values

- kStatus_Success – Start a SAI eDMA send successfully.
- kStatus_InvalidArgument – The input argument is invalid.

status_t SAI_TransferReceiveLoopEDMA(I2S_Type *base, *sai_edma_handle_t* *handle, *sai_transfer_t* *xfer, uint32_t loopTransferCount)

Performs a non-blocking SAI loop transfer using eDMA.

Once the loop transfer start, application can use function SAI_TransferAbortReceiveEDMA to stop the loop transfer.

Note: This function support loop transfer only, such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in *sai_edma_handle_t*, so application could redefine the SAI_XFER_QUEUE_SIZE to determine the proper TCD pool size. This function support one sai channel only.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- xfer – Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (loopTransferCount).
- loopTransferCount – the counts of xfer array.

Return values

- kStatus_Success – Start a SAI eDMA receive successfully.
- kStatus_InvalidArgument – The input argument is invalid.

void SAI_TransferTerminateSendEDMA(I2S_Type *base, *sai_edma_handle_t* *handle)

Terminate all SAI send.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSendEDMA.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

void SAI_TransferTerminateReceiveEDMA(I2S_Type *base, *sai_edma_handle_t* *handle)

Terminate all SAI receive.

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceiveEDMA.

Parameters

- base – SAI base pointer.

- handle – SAI eDMA handle pointer.

`void SAI_TransferAbortSendEDMA(I2S_Type *base, sai_edma_handle_t *handle)`

Aborts a SAI transfer using eDMA.

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call `SAI_TransferTerminateSendEDMA`.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.

`void SAI_TransferAbortReceiveEDMA(I2S_Type *base, sai_edma_handle_t *handle)`

Aborts a SAI receive using eDMA.

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call `SAI_TransferTerminateReceiveEDMA`.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.

`status_t SAI_TransferGetSendCountEDMA(I2S_Type *base, sai_edma_handle_t *handle, size_t *count)`

Gets byte count sent by SAI.

Parameters

- base – SAI base pointer.
- handle – SAI eDMA handle pointer.
- count – Bytes count sent by SAI.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is no non-blocking transaction in progress.

`status_t SAI_TransferGetReceiveCountEDMA(I2S_Type *base, sai_edma_handle_t *handle, size_t *count)`

Gets byte count received by SAI.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.
- count – Bytes count received by SAI.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is no non-blocking transaction in progress.

uint32_t SAI_TransferGetValidTransferSlotsEDMA(I2S_Type *base, sai_edma_handle_t *handle)

Gets valid transfer slot.

This function can be used to query the valid transfer request slot that the application can submit. It should be called in the critical section, that means the application could call it in the corresponding callback function or disable IRQ before calling it in the application, otherwise, the returned value may not correct.

Parameters

- base – SAI base pointer
- handle – SAI eDMA handle pointer.

Return values

valid – slot count that application submit.

FSL_SAI_EDMA_DRIVER_VERSION

Version 2.7.3

enum __sai_edma_interleave

sai interleave type

Values:

enumerator kSAI_EDMAInterleavePerChannelSample

enumerator kSAI_EDMAInterleavePerChannelBlock

typedef struct *sai_edma_handle* sai_edma_handle_t

typedef void (*sai_edma_callback_t)(I2S_Type *base, *sai_edma_handle_t* *handle, status_t status, void *userData)

SAI eDMA transfer callback function for finish and error.

typedef enum *sai_edma_interleave* sai_edma_interleave_t

sai interleave type

MCUX_SDK_SAI_EDMA_RX_ENABLE_INTERNAL

the SAI enable position When calling SAI_TransferReceiveEDMA

MCUX_SDK_SAI_EDMA_TX_ENABLE_INTERNAL

the SAI enable position When calling SAI_TransferSendEDMA

struct sai_edma_handle

#include <fsl_sai_edma.h> SAI DMA transfer handle, users should not touch the content of the handle.

Public Members

edma_handle_t *dmaHandle

DMA handler for SAI send

uint8_t nbytes

eDMA minor byte transfer count initially configured.

uint8_t bytesPerFrame

Bytes in a frame

uint8_t channelMask

Enabled channel mask value, reference *_sai_channel_mask*

`uint8_t channelNums`
total enabled channel nums

`uint8_t channel`
Which data channel

`uint8_t count`
The transfer data count in a DMA request

`uint32_t state`
Internal state for SAI eDMA transfer

`sai_edma_callback_t callback`
Callback for users while transfer finish or error occurs

`void *userData`
User callback parameter

`uint8_t tcd[((4U) + 1U) * sizeof(edma_tcd_t)]`
TCD pool for eDMA transfer.

`sai_transfer_t saiQueue[(4U)]`
Transfer queue storing queued transfer.

`size_t transferSize[(4U)]`
Data bytes need to transfer

`sai_edma_interleave_t interleaveType`
Transfer interleave type

`volatile uint8_t queueUser`
Index for user to queue transfer.

`volatile uint8_t queueDriver`
Index for driver to get the transfer data and size

2.59 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.

2.60 SPDIF: Sony/Philips Digital Interface

`void SPDIF_Init(SPDIF_Type *base, const spdif_config_t *config)`

Initializes the SPDIF peripheral.

Ungates the SPDIF clock, resets the module, and configures SPDIF with a configuration structure. The configuration structure can be custom filled or set with default values by `SPDIF_GetDefaultConfig()`.

Note: This API should be called at the beginning of the application to use the SPDIF driver. Otherwise, accessing the SPDIF module can cause a hard fault because the clock is not enabled.

Parameters

- `base` – SPDIF base pointer
- `config` – SPDIF configuration structure.

`void SPDIF_GetDefaultConfig(spdif_config_t *config)`

Sets the SPDIF configuration structure to default values.

This API initializes the configuration structure for use in `SPDIF_Init`. The initialized structure can remain unchanged in `SPDIF_Init`, or it can be modified before calling `SPDIF_Init`. This is an example.

```
spdif_config_t config;
SPDIF_GetDefaultConfig(&config);
```

Parameters

- `config` – pointer to master configuration structure

`void SPDIF_Deinit(SPDIF_Type *base)`

De-initializes the SPDIF peripheral.

This API gates the SPDIF clock. The SPDIF module can't operate unless `SPDIF_Init` is called to enable the clock.

Parameters

- `base` – SPDIF base pointer

`uint32_t SPDIF_GetInstance(SPDIF_Type *base)`

Get the instance number for SPDIF.

Parameters

- `base` – SPDIF base pointer.

`static inline void SPDIF_TxFIFOReset(SPDIF_Type *base)`

Resets the SPDIF Tx.

This function makes Tx FIFO in reset mode.

Parameters

- `base` – SPDIF base pointer

`static inline void SPDIF_RxFIFOReset(SPDIF_Type *base)`

Resets the SPDIF Rx.

This function enables the software reset and FIFO reset of SPDIF Rx. After reset, clear the reset bit.

Parameters

- base – SPDIF base pointer

void SPDIF_TxEnable(SPDIF_Type *base, bool enable)

Enables/disables the SPDIF Tx.

Parameters

- base – SPDIF base pointer
- enable – True means enable SPDIF Tx, false means disable.

static inline void SPDIF_RxEnable(SPDIF_Type *base, bool enable)

Enables/disables the SPDIF Rx.

Parameters

- base – SPDIF base pointer
- enable – True means enable SPDIF Rx, false means disable.

static inline uint32_t SPDIF_GetStatusFlag(SPDIF_Type *base)

Gets the SPDIF status flag state.

Parameters

- base – SPDIF base pointer

Returns

SPDIF status flag value. Use the `_spdif_interrupt_enable_t` to get the status value needed.

static inline void SPDIF_ClearStatusFlags(SPDIF_Type *base, uint32_t mask)

Clears the SPDIF status flag state.

Parameters

- base – SPDIF base pointer
- mask – State mask. It can be a combination of the `_spdif_interrupt_enable_t` member. Notice these members cannot be included, as these flags cannot be cleared by writing 1 to these bits:
 - kSPDIF_UChannelReceiveRegisterFull
 - kSPDIF_QChannelReceiveRegisterFull
 - kSPDIF_TxFIFOEmpty
 - kSPDIF_RxFIFOFull

static inline void SPDIF_EnableInterrupts(SPDIF_Type *base, uint32_t mask)

Enables the SPDIF Tx interrupt requests.

Parameters

- base – SPDIF base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSPDIF_WordStartInterruptEnable
 - kSPDIF_SyncErrorInterruptEnable
 - kSPDIF_FIFOWarningInterruptEnable
 - kSPDIF_FIFORequestInterruptEnable
 - kSPDIF_FIFOErrorInterruptEnable

```
static inline void SPDIF__DisableInterrupts(SPDIF_Type *base, uint32_t mask)
```

Disables the SPDIF Tx interrupt requests.

Parameters

- base – SPDIF base pointer
- mask – interrupt source The parameter can be a combination of the following sources if defined.
 - kSPDIF_WordStartInterruptEnable
 - kSPDIF_SyncErrorInterruptEnable
 - kSPDIF_FIFOWarningInterruptEnable
 - kSPDIF_FIFORequestInterruptEnable
 - kSPDIF_FIFOErrorInterruptEnable

```
static inline void SPDIF__EnableDMA(SPDIF_Type *base, uint32_t mask, bool enable)
```

Enables/disables the SPDIF DMA requests.

Parameters

- base – SPDIF base pointer
- mask – SPDIF DMA enable mask, The parameter can be a combination of the following sources if defined
 - kSPDIF_RxDMAEnable
 - kSPDIF_TxDMAEnable
- enable – True means enable DMA, false means disable DMA.

```
static inline uint32_t SPDIF__TxGetLeftDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Tx left data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF__TxGetRightDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Tx right data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF__RxGetLeftDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Rx left data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
static inline uint32_t SPDIF_RxGetRightDataRegisterAddress(SPDIF_Type *base)
```

Gets the SPDIF Rx right data register address.

This API is used to provide a transfer address for the SPDIF DMA transfer configuration.

Parameters

- base – SPDIF base pointer.

Returns

data register address.

```
void SPDIF_TxSetSampleRate(SPDIF_Type *base, uint32_t sampleRate_Hz, uint32_t  
                           sourceClockFreq_Hz)
```

Configures the SPDIF Tx sample rate.

The audio format can be changed at run-time. This function configures the sample rate.

Parameters

- base – SPDIF base pointer.
- sampleRate_Hz – SPDIF sample rate frequency in Hz.
- sourceClockFreq_Hz – SPDIF tx clock source frequency in Hz.

```
uint32_t SPDIF_GetRxSampleRate(SPDIF_Type *base, uint32_t clockSourceFreq_Hz)
```

Configures the SPDIF Rx audio format.

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – SPDIF base pointer.
- clockSourceFreq_Hz – SPDIF system clock frequency in hz.

```
void SPDIF_WriteBlocking(SPDIF_Type *base, uint8_t *buffer, uint32_t size)
```

Sends data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SPDIF base pointer.
- buffer – Pointer to the data to be written.
- size – Bytes to be written.

```
static inline void SPDIF_WriteLeftData(SPDIF_Type *base, uint32_t data)
```

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

```
static inline void SPDIF_WriteRightData(SPDIF_Type *base, uint32_t data)
```

Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

static inline void SPDIF_WriteChannelStatusHigh(SPDIF_Type *base, uint32_t data)
Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

static inline void SPDIF_WriteChannelStatusLow(SPDIF_Type *base, uint32_t data)
Writes data into SPDIF FIFO.

Parameters

- base – SPDIF base pointer.
- data – Data needs to be written.

void SPDIF_ReadBlocking(SPDIF_Type *base, uint8_t *buffer, uint32_t size)
Receives data using a blocking method.

Note: This function blocks by polling until data is ready to be sent.

Parameters

- base – SPDIF base pointer.
- buffer – Pointer to the data to be read.
- size – Bytes to be read.

static inline uint32_t SPDIF_ReadLeftData(SPDIF_Type *base)
Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

static inline uint32_t SPDIF_ReadRightData(SPDIF_Type *base)
Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

static inline uint32_t SPDIF_ReadChannelStatusHigh(SPDIF_Type *base)
Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

static inline uint32_t SPDIF_ReadChannelStatusLow(SPDIF_Type *base)
Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

```
static inline uint32_t SPDIF_ReadQChannel(SPDIF_Type *base)
```

Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

```
static inline uint32_t SPDIF_ReadUChannel(SPDIF_Type *base)
```

Reads data from the SPDIF FIFO.

Parameters

- base – SPDIF base pointer.

Returns

Data in SPDIF FIFO.

```
void SPDIF_TransferTxCreateHandle(SPDIF_Type *base, spdif_handle_t *handle,  
                                spdif_transfer_callback_t callback, void *userData)
```

Initializes the SPDIF Tx handle.

This function initializes the Tx handle for the SPDIF Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SPDIF base pointer
- handle – SPDIF handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function

```
void SPDIF_TransferRxCreateHandle(SPDIF_Type *base, spdif_handle_t *handle,  
                                spdif_transfer_callback_t callback, void *userData)
```

Initializes the SPDIF Rx handle.

This function initializes the Rx handle for the SPDIF Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

- base – SPDIF base pointer.
- handle – SPDIF handle pointer.
- callback – Pointer to the user callback function.
- userData – User parameter passed to the callback function.

```
status_t SPDIF_TransferSendNonBlocking(SPDIF_Type *base, spdif_handle_t *handle,  
                                       spdif_transfer_t *xfer)
```

Performs an interrupt non-blocking send transfer on SPDIF.

Note: This API returns immediately after the transfer initiates. Call the `SPDIF_TxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SPDIF_Busy`, the transfer is finished.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the `spdif_handle_t` structure which stores the transfer state.
- xfer – Pointer to the `spdif_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully started the data receive.
- `kStatus_SPDIF_TxBusy` – Previous receive still not finished.
- `kStatus_InvalidArgument` – The input parameter is invalid.

status_t SPDIF_TransferReceiveNonBlocking(*SPDIF_Type* *base, *spdif_handle_t* *handle, *spdif_transfer_t* *xfer)

Performs an interrupt non-blocking receive transfer on SPDIF.

Note: This API returns immediately after the transfer initiates. Call the `SPDIF_RxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SPDIF_Busy`, the transfer is finished.

Parameters

- base – SPDIF base pointer
- handle – Pointer to the `spdif_handle_t` structure which stores the transfer state.
- xfer – Pointer to the `spdif_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully started the data receive.
- `kStatus_SPDIF_RxBusy` – Previous receive still not finished.
- `kStatus_InvalidArgument` – The input parameter is invalid.

status_t SPDIF_TransferGetSendCount(*SPDIF_Type* *base, *spdif_handle_t* *handle, *size_t* *count)

Gets a set byte count.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the `spdif_handle_t` structure which stores the transfer state.
- count – Bytes count sent.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

status_t SPDIF_TransferGetReceiveCount(*SPDIF_Type* *base, *spdif_handle_t* *handle, *size_t* *count)

Gets a received byte count.

Parameters

- base – SPDIF base pointer.
- handle – Pointer to the `spdif_handle_t` structure which stores the transfer state.

- `count` – Bytes count received.

Return values

- `kStatus_Success` – Succeed get the transfer count.
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`void SPDIF_TransferAbortSend(SPDIF_Type *base, spdif_handle_t *handle)`

Aborts the current send.

Note: This API can be called any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – SPDIF base pointer.
- `handle` – Pointer to the `spdif_handle_t` structure which stores the transfer state.

`void SPDIF_TransferAbortReceive(SPDIF_Type *base, spdif_handle_t *handle)`

Aborts the current IRQ receive.

Note: This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – SPDIF base pointer
- `handle` – Pointer to the `spdif_handle_t` structure which stores the transfer state.

`void SPDIF_TransferTxHandleIRQ(SPDIF_Type *base, spdif_handle_t *handle)`

Tx interrupt handler.

Parameters

- `base` – SPDIF base pointer.
- `handle` – Pointer to the `spdif_handle_t` structure.

`void SPDIF_TransferRxHandleIRQ(SPDIF_Type *base, spdif_handle_t *handle)`

Rx interrupt handler.

Parameters

- `base` – SPDIF base pointer.
- `handle` – Pointer to the `spdif_handle_t` structure.

FSL_SPDIF_DRIVER_VERSION

Version 2.0.7

SPDIF return status.

Values:

enumerator `kStatus_SPDIF_RxDPLLLocked`

SPDIF Rx PLL locked.

enumerator kStatus_SPDIF_TxFIFOError

SPDIF Tx FIFO error.

enumerator kStatus_SPDIF_TxFIFOResync

SPDIF Tx left and right FIFO resync.

enumerator kStatus_SPDIF_RxCnew

SPDIF Rx status channel value updated.

enumerator kStatus_SPDIF_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 flag 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 flag 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

2.61 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

2.62 TMU: Thermal Management Unit Driver

`enum __tmu_monitor_site`
TMU monitor site.

Values:

enumerator kTMU_MonitorSite0

TMU monitoring site 0.

enum _tmu_interrupt_enable

TMU interrupt enable.

Values:

enumerator kTMU_ImmediateTemperatureInterruptEnable

Immediate temperature threshold exceeded interrupt enable.

enumerator kTMU_AverageTemperatureInterruptEnable

Average temperature threshold exceeded interrupt enable.

enumerator kTMU_AverageTemperatureCriticalInterruptEnable

Average temperature critical threshold exceeded interrupt enable.

enumerator kTMU_RisingTemperatureCriticalInterruptEnable

Rising temperature critical threshold exceeded interrupt enable.

enumerator kTMU_FallingTemperatureCriticalInterruptEnable

Falling temperature critical threshold exceeded interrupt enable.

enum _tmu_interrupt_status_flags

TMU interrupt status flags.

Values:

enumerator kTMU_ImmediateTemperatureStatusFlags

Immediate temperature threshold exceeded(IHTT).

enumerator kTMU_AverageTemperatureStatusFlags

Average temperature threshold exceeded(AHTT).

enumerator kTMU_AverageTemperatureCriticalStatusFlags

Average temperature critical threshold exceeded.(AHTCT)

enumerator kTMU_RisingTemperatureCriticalStatusFlags

Rising temperature critical threshold exceeded.(RTRCT)

enumerator kTMU_FallingTemperatureCriticalStatusFlags

Falling temperature critical threshold exceeded.(FTRCT)

enum _tmu_status_flags

TMU status flags.

Values:

enumerator kTMU_IntervalExceededStatusFlags

Monitoring interval exceeded. The time required to perform measurement of all monitored sites has exceeded the monitoring interval as defined by TMTMIR.

enumerator kTMU_OutOfLowRangeStatusFlags

Out-of-range low temperature measurement detected. A temperature sensor detected a temperature reading below the lowest measurable temperature of 0 °C.

enumerator kTMU_OutOfHighRangeStatusFlags

Out-of-range high temperature measurement detected. A temperature sensor detected a temperature reading above the highest measurable temperature of 160 °C.

enum _tmu_average_low_pass_filter

Average low pass filter setting.

Values:

enumerator kTMU_AverageLowPassFilter1_0

Average low pass filter = 1.

enumerator kTMU_AverageLowPassFilter0_5

Average low pass filter = 0.5.

enumerator kTMU_AverageLowPassFilter0_25

Average low pass filter = 0.25.

enumerator kTMU_AverageLowPassFilter0_125

Average low pass filter = 0.125.

typedef struct *tmu_threshold_config* tmu_threshold_config_t

configuration for TMU threshold.

typedef struct *tmu_interrupt_status* tmu_interrupt_status_t

TMU interrupt status.

typedef enum *tmu_average_low_pass_filter* tmu_average_low_pass_filter_t

Average low pass filter setting.

typedef struct *tmu_config* tmu_config_t

Configuration for TMU module.

void TMU_Init(TMU_Type *base, const *tmu_config_t* *config)

Enable the access to TMU registers and Initialize TMU module.

Parameters

- base – TMU peripheral base address.
- config – Pointer to configuration structure. Refer to “tmu_config_t” structure.

void TMU_Deinit(TMU_Type *base)

De-initialize TMU module and Disable the access to DCDC registers.

Parameters

- base – TMU peripheral base address.

void TMU_GetDefaultConfig(*tmu_config_t* *config)

Gets the default configuration for TMU.

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

Example:

```
config->monitorInterval = 0U;
config->monitorSiteSelection = 0U;
config->averageLPF = kTMU_AverageLowPassFilter1_0;
```

Parameters

- config – Pointer to TMU configuration structure.

static inline void TMU_Enable(TMU_Type *base, bool enable)

Enable/Disable the TMU module.

Parameters

- base – TMU peripheral base address.
- enable – Switcher to enable/disable TMU.

static inline void TMU_EnableInterrupts(TMU_Type *base, uint32_t mask)

Enable the TMU interrupts.

Parameters

- base – TMU peripheral base address.
- mask – The interrupt mask. Refer to “_tmu_interrupt_enable” enumeration.

static inline void TMU_DisableInterrupts(TMU_Type *base, uint32_t mask)

Disable the TMU interrupts.

Parameters

- base – TMU peripheral base address.
- mask – The interrupt mask. Refer to “_tmu_interrupt_enable” enumeration.

void TMU_GetInterruptStatusFlags(TMU_Type *base, *tmu_interrupt_status_t* *status)

Get interrupt status flags.

Parameters

- base – TMU peripheral base address.
- status – The pointer to interrupt status structure. Record the current interrupt status. Please refer to “tmu_interrupt_status_t” structure.

void TMU_ClearInterruptStatusFlags(TMU_Type *base, uint32_t mask)

Clear interrupt status flags and corresponding interrupt critical site capture register.

Parameters

- base – TMU peripheral base address.
- mask – The mask of interrupt status flags. Refer to “_tmu_interrupt_status_flags” enumeration.

static inline uint32_t TMU_GetStatusFlags(TMU_Type *base)

Get TMU status flags.

Parameters

- base – TMU peripheral base address.

Returns

The mask of status flags. Refer to “_tmu_status_flags” enumeration.

status_t TMU_GetHighestTemperature(TMU_Type *base, uint32_t *temperature)

Get the highest temperature reached for any enabled monitored site within the temperature sensor range.

Parameters

- base – TMU peripheral base address.
- temperature – Highest temperature recorded in degrees Celsius by any enabled monitored site.

Return values

- kStatus_Success – Temperature reading is valid.
- kStatus_Fail – Temperature reading is not valid due to no measured temperature within the sensor range of 0-160 °C for an enabled monitored site.

Returns

Execution status.

status_t TMU_GetLowestTemperature(TMU_Type *base, uint32_t *temperature)

Get the lowest temperature reached for any enabled monitored site within the temperature sensor range.

Parameters

- base – TMU peripheral base address.
- temperature – Lowest temperature recorded in degrees Celsius by any enabled monitored site.

Return values

- kStatus_Success – Temperature reading is valid.
- kStatus_Fail – Temperature reading is not valid due to no measured temperature within the sensor range of 0-160 °C for an enabled monitored site.

Returns

Execution status.

status_t TMU_GetImmediateTemperature(TMU_Type *base, float *temperature)

Get the last immediate temperature at site n. The site must be part of the list of enabled monitored sites as defined by monitorSiteSelection in “tmu_config_t” structure.

Parameters

- base – TMU peripheral base address.
- temperature – Last immediate temperature reading at site 0.

Return values

- kStatus_Success – Temperature reading is valid.
- kStatus_Fail – Temperature reading is not valid because temperature out of sensor range or first measurement still pending.

Returns

Execution status.

status_t TMU_GetAverageTemperature(TMU_Type *base, uint32_t *temperature)

Get the last average temperature at site n. The site must be part of the list of enabled monitored sites as defined by monitorSiteSelection in “tmu_config_t” structure.

Parameters

- base – TMU peripheral base address.
- temperature – Last average temperature reading at site 0.

Return values

- kStatus_Success – Temperature reading is valid.
- kStatus_Fail – Temperature reading is not valid because temperature out of sensor range or first measurement still pending.

Returns

Execution status.

void TMU_SetHighTemperatureThresold(TMU_Type *base, const *tmu_thresold_config_t* *config)

Configure the high temperature thresold value and enable/disable relevant thresold.

Parameters

- base – TMU peripheral base address.

- `config` – Pointer to configuration structure. Refer to “`tmu_threshold_config_t`” structure.

`FSL_TMU_DRIVER_VERSION`

TMU driver version.

Version 2.1.0.

`struct _tmu_threshold_config`

`#include <fsl_tmu.h>` configuration for TMU threshold.

Public Members

`bool immediateThresoldEnable`

Enable high temperature immediate threshold.

`bool averageThresoldEnable`

Enable high temperature average threshold.

`bool averageCriticalThresoldEnable`

Enable high temperature average critical threshold.

`bool risingCriticalThresoldEnable`

Enable rising temperature rate critical threshold.

`bool fallingCriticalThresoldEnable`

Enable rising temperature rate critical threshold.

`uint8_t immediateThresoldValue`

Range:0U-160U. Valid when corresponding threshold is enabled. High temperature immediate threshold value. Determines the current upper temperature threshold, for anyenabled monitored site.

`uint8_t averageThresoldValue`

Range:0U-160U. Valid when corresponding threshold is enabled. High temperature average threshold value. Determines the average upper temperature threshold, for any enabled monitored site.

`uint8_t averageCriticalThresoldValue`

Range:0U-160U. Valid when corresponding threshold is enabled. High temperature average critical threshold value. Determines the average upper critical temperature threshold, for any enabled monitored site.

`uint8_t risingfallingCriticalThresoldValue`

Range:0U-160U. Valid when corresponding threshold is enabled. Rising temperature rate critical threshold value. Determines the rising upper critical temperature threshold, for any enabled monitored site.

`struct _tmu_interrupt_status`

`#include <fsl_tmu.h>` TMU interrupt status.

Public Members

`uint32_t interruptDetectMask`

The mask of interrupt status flags. Refer to “`_tmu_interrupt_status_flags`” enumeration.

`uint16_t immediateInterruptsSiteMask`

The mask of the temperature sensor site associated with a detected IHTT event. Please refer to “`_tmu_monitor_site`” enumeration.

uint16_t averageInterruptsSiteMask

The mask of the temperature sensor site associated with a detected AHTT event. Please refer to “_tmu_monitor_site” enumeration.

uint16_t averageCriticalInterruptsSiteMask

The mask of the temperature sensor site associated with a detected AHTCT event. Please refer to “_tmu_monitor_site” enumeration.

uint16_t risingCriticalInterruptsSiteMask

The mask of the temperature sensor site associated with a detected RTRCT event. Please refer to “_tmu_monitor_site” enumeration.

uint16_t fallingCriticalInterruptsSiteMask

The mask of the temperature sensor site associated with a detected FTRCT event. Please refer to “_tmu_monitor_site” enumeration.

struct __tmu_config

#include <fsl_tmu.h> Configuration for TMU module.

Public Members

uint8_t monitorInterval

Temperature monitoring interval in seconds. Please refer to specific table in RM.

uint16_t monitorSiteSelection

By setting the select bit for a temperature sensor site, it is enabled and included in all monitoring functions. If no site is selected, site 0 is monitored by default. Refer to “_tmu_monitor_site” enumeration. Please look up relevant table in reference manual.

tmu_average_low_pass_filter_t averageLPF

The average temperature is calculated as: $ALPF \times \text{Current\_Temp} + (1 - ALPF) \times \text{Average\_Temp}$. For proper operation, this field should only change when monitoring is disabled.

2.63 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_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 channel CnV register value.

static inline uint32_t TPM_GetStatusFlags(TPM_Type *base)

Gets the TPM status flags.

Parameters

- base – TPM peripheral base address

Returns

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

static inline void TPM_ClearStatusFlags(TPM_Type *base, uint32_t mask)

Clears the TPM status flags.

Parameters

- base – TPM peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `tpm_status_flags_t`

static inline *status_t* TPM_SetTimerPeriod(TPM_Type *base, uint32_t ticks)

Sets the timer period in units of ticks.

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note:

- This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.
 - Call the utility macros provided in the `fsl_common.h` to convert usec or msec to ticks.
-

Parameters

- base – TPM peripheral base address
- ticks – A timer period in units of ticks, which should be equal or greater than 1.

Returns

`kStatus_Success` PWM setup successful
`kStatus_Timeout` PWM setup timeout when write register `CnSC`

static inline uint32_t TPM_GetCurrentTimerCount(TPM_Type *base)

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – TPM peripheral base address

Returns

The current counter value in ticks

static inline void TPM_StartTimer(TPM_Type *base, *tpm_clock_source_t* clockSource)

Starts the TPM counter.

Parameters

- base – TPM peripheral base address
- clockSource – TPM clock source; once clock source is set the counter will start running

static inline *status_t* TPM_StopTimer(TPM_Type *base)

Stops the TPM counter.

Parameters

- base – TPM peripheral base address

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout
when write register CnSC

FSL_TPM_DRIVER_VERSION

TPM driver version 2.4.0.

enum *_tpm_chnl*

List of TPM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kTPM_Chnl_0
TPM channel number 0

enumerator kTPM_Chnl_1
TPM channel number 1

enumerator kTPM_Chnl_2
TPM channel number 2

enumerator kTPM_Chnl_3
TPM channel number 3

enumerator kTPM_Chnl_4
TPM channel number 4

enumerator kTPM_Chnl_5
TPM channel number 5

enumerator kTPM_Chnl_6
TPM channel number 6

enumerator kTPM_Chnl_7
TPM channel number 7

enum *_tpm_pwm_mode*

TPM PWM operation modes.

Values:

enumerator kTPM_EdgeAlignedPwm
Edge aligned PWM

enumerator kTPM_CenterAlignedPwm
Center aligned PWM

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_NoPwmSignal

No PWM output on pin

enumerator kTPM_LowTrue

Low true pulses

enumerator kTPM_HighTrue

High true pulses

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_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_clock_source
TPM clock source selection.

Values:

enumerator kTPM_SystemClock
System clock
enumerator kTPM_ExternalClock
External TPM_EXTCLK pin 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_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_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 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.

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_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)

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

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

2.64 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.

`FSL_TRDC_DRIVER_VERSION`

2.65 Trdc_core

`typedef struct _TRDC_General_Type TRDC_General_Type`

TRDC general configuration register definition.

`typedef struct _TRDC_FLW_Type TRDC_FLW_Type`

TRDC flash logical control register definition.

`typedef struct _TRDC_DomainError_Type TRDC_DomainError_Type`

TRDC domain error register definition.

`typedef struct _TRDC_DomainAssignment_Type TRDC_DomainAssignment_Type`

TRDC master domain assignment register definition.

`typedef struct _TRDC_MBC_Type TRDC_MBC_Type`

TRDC MBC control register definition.

`typedef struct _TRDC_MRC_Type TRDC_MRC_Type`

TRDC MRC control register definition. `MRC_DOM0_RGD_W[region][word]`.

`TRDC_GENERAL_BASE(base)`

TRDC base address convert macro.

`TRDC_FLW_BASE(base)`

`TRDC_DOMAIN_ERROR_BASE(base)`

`TRDC_DOMAIN_ASSIGNMENT_BASE(base)`

`TRDC_MBC_BASE(base, instance)`

`TRDC_MRC_BASE(base, instance)`

`struct _TRDC_General_Type`

`#include <fsl_trdc_core.h>` TRDC general configuration register definition.

Public Members

`__IO uint32_t TRDC_CR`

TRDC Register, offset: 0x0

`__I uint32_t TRDC_HWCFG0`

TRDC Hardware Configuration Register 0, offset: 0xF0

`__I uint32_t TRDC_HWCFG1`

TRDC Hardware Configuration Register 1, offset: 0xF4

`__I uint32_t TRDC_HWCFG2`

TRDC Hardware Configuration Register 2, offset: 0xF8


```

__IO uint32_t TRDC_HWCFG3
    TRDC Hardware Configuration Register 3, offset: 0xFC

__IO uint8_t DACFG [8]
    Domain Assignment Configuration Register, array offset: 0x100, array step: 0x1

__IO uint32_t TRDC_IDAU_CR
    TRDC IDAU Control Register, offset: 0x1C0

struct __TRDC_FLW_Type
    #include <fsl_trdc_core.h> TRDC flash logical control register definition.

```

Public Members

```

__IO uint32_t TRDC_FLW_CTL
    TRDC FLW Control, offset: 0x1E0

__IO uint32_t TRDC_FLW_PBASE
    TRDC FLW Physical Base, offset: 0x1E4

__IO uint32_t TRDC_FLW_ABASE
    TRDC FLW Array Base, offset: 0x1E8

__IO uint32_t TRDC_FLW_BCNT
    TRDC FLW Block Count, offset: 0x1EC

struct __TRDC_DomainError_Type
    #include <fsl_trdc_core.h> TRDC domain error register definition.

```

Public Members

```

__IO uint32_t TRDC_FDID
    TRDC Fault Domain ID, offset: 0x1FC

__IO uint32_t TRDC_DERRLOC [16]
    TRDC Domain Error Location Register, array offset: 0x200, array step: 0x4

struct __TRDC_DomainAssignment_Type
    #include <fsl_trdc_core.h> TRDC master domain assignment register definition.

```

Public Members

```

__IO uint32_t PID [8]
    Process Identifier, array offset: 0x700, array step: 0x4

struct __TRDC_MBC_Type
    #include <fsl_trdc_core.h> TRDC MBC control register definition.

```

Public Members

```

__IO uint32_t MBC_MEM_GLBCFG [4]
    MBC Global Configuration Register, array offset: 0x10000, array step: index*0x2000,
    index2*0x4

__IO uint32_t MBC_NSE_BLK_INDEX
    MBC NonSecure Enable Block Index, array offset: 0x10010, array step: 0x2000

```

- ___O uint32_t MBC_NSE_BLK_SET
MBC NonSecure Enable Block Set, array offset: 0x10014, array step: 0x2000
- ___O uint32_t MBC_NSE_BLK_CLR
MBC NonSecure Enable Block Clear, array offset: 0x10018, array step: 0x2000
- ___O uint32_t MBC_NSE_BLK_CLR_ALL
MBC NonSecure Enable Block Clear All, array offset: 0x1001C, array step: 0x2000
- ___IO uint32_t MBC_MEMN_GLBAC [8]
MBC Global Access Control, array offset: 0x10020, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10040, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10140, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10180, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x101A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x101A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x101C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x101D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM0_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x101F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10240, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10340, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10380, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x103A0, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM1_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x103A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x103C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x103D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM1_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x103F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10440, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10540, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10580, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x105A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x105A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x105C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x105D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM2_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x105F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10640, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10740, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10780, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM3_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x107A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x107A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x107C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x107D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM3_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x107F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10840, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10940, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10980, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x109A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x109A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x109C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x109D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM4_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x109F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10A40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10B40, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM5_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10B80, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10BA0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10BA8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10BC8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10BD0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM5_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10BF0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10C40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10D40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10D80, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10DA0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10DA8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10DC8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10DD0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM6_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10DF0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x10E40, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM7_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x10F40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10F80, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10FA0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10FA8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10FC8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x10FD0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM7_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x10FF0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11040, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11140, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11180, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x111A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x111A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x111C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x111D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM8_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x111F0, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM9_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11240, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11340, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11380, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x113A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x113A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x113C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x113D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM9_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x113F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11440, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11540, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11580, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x115A0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x115A8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x115C8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM10_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x115D0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM10_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x115F0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11640, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11740, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11780, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x117A0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x117A8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x117C8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x117D0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM11_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x117F0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11840, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11940, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11980, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x119A0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x119A8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM12_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x119C8, array step: index*0x2000, index2*0x4

- ___IO uint32_t MBC_DOM12_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x119D0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM12_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x119F0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11A40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11B40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11B80, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x11BA0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11BA8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM2_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x11BC8, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM3_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11BD0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM13_MEM3_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x11BF0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM14_MEM0_BLK_CFG_W [64]
MBC Memory Block Configuration Word, array offset: 0x11C40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM14_MEM0_BLK_NSE_W [16]
MBC Memory Block NonSecure Enable Word, array offset: 0x11D40, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM14_MEM1_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11D80, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM14_MEM1_BLK_NSE_W [2]
MBC Memory Block NonSecure Enable Word, array offset: 0x11DA0, array step: index*0x2000, index2*0x4
- ___IO uint32_t MBC_DOM14_MEM2_BLK_CFG_W [8]
MBC Memory Block Configuration Word, array offset: 0x11DA8, array step: index*0x2000, index2*0x4

```
__IO uint32_t MBC_DOM14_MEM2_BLK_NSE_W [2]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11DC8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM14_MEM3_BLK_CFG_W [8]
    MBC Memory Block Configuration Word, array offset: 0x11DD0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM14_MEM3_BLK_NSE_W [2]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11DF0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM0_BLK_CFG_W [64]
    MBC Memory Block Configuration Word, array offset: 0x11E40, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM0_BLK_NSE_W [16]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11F40, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM1_BLK_CFG_W [8]
    MBC Memory Block Configuration Word, array offset: 0x11F80, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM1_BLK_NSE_W [2]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11FA0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM2_BLK_CFG_W [8]
    MBC Memory Block Configuration Word, array offset: 0x11FA8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM2_BLK_NSE_W [2]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11FC8, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM3_BLK_CFG_W [8]
    MBC Memory Block Configuration Word, array offset: 0x11FD0, array step: index*0x2000, index2*0x4

__IO uint32_t MBC_DOM15_MEM3_BLK_NSE_W [2]
    MBC Memory Block NonSecure Enable Word, array offset: 0x11FF0, array step: index*0x2000, index2*0x4
```

struct __TRDC_MRC_Type

```
#include <fsl_trdc_core.h>    TRDC    MRC    control    register    definition.
MRC_DOM0_RGD_W[region][word].
```

Public Members

```
__IO uint32_t MRC_GLBCFG
    MRC Global Configuration Register, array offset: 0x14000, array step: 0x1000

__IO uint32_t MRC_NSE_RGN_INDIRECT
    MRC NonSecure Enable Region Indirect, array offset: 0x14010, array step: 0x1000

__IO uint32_t MRC_NSE_RGN_SET
    MRC NonSecure Enable Region Set, array offset: 0x14014, array step: 0x1000
```

```

__O uint32_t MRC_NSE_RGN_CLR
    MRC NonSecure Enable Region Clear, array offset: 0x14018, array step: 0x1000
__O uint32_t MRC_NSE_RGN_CLR_ALL
    MRC NonSecure Enable Region Clear All, array offset: 0x1401C, array step: 0x1000
__IO uint32_t MRC_GLBAC [8]
    MRC Global Access Control, array offset: 0x14020, array step: index*0x1000, index2*0x4
__IO uint32_t MRC_DOM0_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14040,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM0_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x140C0, array step: 0x1000
__IO uint32_t MRC_DOM1_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14140,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM1_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x141C0, array step: 0x1000
__IO uint32_t MRC_DOM2_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14240,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM2_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x142C0, array step: 0x1000
__IO uint32_t MRC_DOM3_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14340,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM3_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x143C0, array step: 0x1000
__IO uint32_t MRC_DOM4_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14440,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM4_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x144C0, array step: 0x1000
__IO uint32_t MRC_DOM5_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14540,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM5_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x145C0, array step: 0x1000
__IO uint32_t MRC_DOM6_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14640,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM6_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x146C0, array step: 0x1000
__IO uint32_t MRC_DOM7_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14740,
    array step: index*0x1000, index2*0x8, index3*0x4

```

```
__IO uint32_t MRC_DOM7_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x147C0, array step: 0x1000
__IO uint32_t MRC_DOM8_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14840,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM8_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x148C0, array step: 0x1000
__IO uint32_t MRC_DOM9_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14940,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM9_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x149C0, array step: 0x1000
__IO uint32_t MRC_DOM10_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14A40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM10_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14AC0, array step: 0x1000
__IO uint32_t MRC_DOM11_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14B40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM11_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14BC0, array step: 0x1000
__IO uint32_t MRC_DOM12_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14C40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM12_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14CC0, array step: 0x1000
__IO uint32_t MRC_DOM13_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14D40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM13_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14DC0, array step: 0x1000
__IO uint32_t MRC_DOM14_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14E40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM14_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14EC0, array step: 0x1000
__IO uint32_t MRC_DOM15_RGD_W [16][2]
    MRC Region Descriptor Word 0..MRC Region Descriptor Word 1, array offset: 0x14F40,
    array step: index*0x1000, index2*0x8, index3*0x4
__IO uint32_t MRC_DOM15_RGD_NSE
    MRC Region Descriptor NonSecure Enable, array offset: 0x14FC0, array step: 0x1000
struct MBC_DERR
```

Public Members

```
___I uint32_t W0
    MBC Domain Error Word0 Register, array offset: 0x400, array step: 0x10
___I uint32_t W1
    MBC Domain Error Word1 Register, array offset: 0x404, array step: 0x10
___O uint32_t W3
    MBC Domain Error Word3 Register, array offset: 0x40C, array step: 0x10
struct MRC_DERR
```

Public Members

```
___I uint32_t W0
    MRC Domain Error Word0 Register, array offset: 0x480, array step: 0x10
___I uint32_t W1
    MRC Domain Error Word1 Register, array offset: 0x484, array step: 0x10
___O uint32_t W3
    MRC Domain Error Word3 Register, array offset: 0x48C, array step: 0x10
union ___unnamed78___
```

Public Members

```
struct _TRDC_DomainAssignment_Type MDA_DFMT0[8]
struct _TRDC_DomainAssignment_Type MDA_DFMT1[8]
struct MDA_DFMT0
```

Public Members

```
___IO uint32_t MDA_W_DFMT0 [8]
    DAC Master Domain Assignment Register, array offset: 0x800, array step: index*0x20,
    index2*0x4
struct MDA_DFMT1
```

Public Members

```
___IO uint32_t MDA_W_DFMT1 [1]
    DAC Master Domain Assignment Register, array offset: 0x800, array step: index*0x20,
    index2*0x4
```

2.66 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.

2.67 TSTMR: Timestamp Timer Driver

`void TSTMR_Init(TSTMR_Type *base)`

Init TSTMR.

This function initializes the TSTMR module.

Parameters

- `base` – TSTMR peripheral base address.

`void TSTMR_Deinit(TSTMR_Type *base)`

Deinit TSTMR.

This function deinitializes the TSTMR module.

Parameters

- `base` – TSTMR peripheral base address.

`FSL_TSTMR_DRIVER_VERSION`

Version 2.1.0

`static inline uint64_t TSTMR_ReadTimeStamp(TSTMR_Type *base)`

Reads the time stamp.

This function reads the low and high registers and returns the 56-bit free running counter value. This can be read by software at any time to determine the software ticks. TSTMR registers can be read with 32-bit accesses only. The TSTMR LOW read should occur first, followed by the TSTMR HIGH read.

Parameters

- `base` – TSTMR peripheral base address.

Returns

The 56-bit time stamp value.

`void TSTMR_DelayUs(TSTMR_Type *base, uint64_t delayInUs)`

Delays for a specified number of microseconds.

This function repeatedly reads the timestamp register and waits for the user-specified delay value.

Parameters

- `base` – TSTMR peripheral base address.
- `delayInUs` – Delay value in microseconds.

2.68 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 void USDHC__SelectVoltage(USDHC_Type *base, bool en18v)

Selects the USDHC output voltage.

Parameters

- base – USDHC peripheral base address.
- en18v – True means 1.8V, false means 3.0V.

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

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_SupportV180Flag
Support voltage 1.8V.

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.

uint8_t readBurstLen

Read burst len.

uint8_t writeBurstLen

Write burst len.

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.

usdhc_burst_len_t burstLen
Burst len config.

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.

2.69 WDOG32: 32-bit Watchdog Timer

`void WDOG32_GetDefaultConfig(wdog32_config_t *config)`

Initializes the WDOG32 configuration structure.

This function initializes the WDOG32 configuration structure to default values. The default values are:

```
wdog32Config->enableWdog32 = true;
wdog32Config->clockSource = kWDOG32_ClockSource1;
wdog32Config->prescaler = kWDOG32_ClockPrescalerDivide1;
wdog32Config->workMode.enableWait = true;
wdog32Config->workMode.enableStop = false;
wdog32Config->workMode.enableDebug = false;
wdog32Config->testMode = kWDOG32_TestModeDisabled;
wdog32Config->enableUpdate = true;
wdog32Config->enableInterrupt = false;
wdog32Config->enableWindowMode = false;
wdog32Config->windowValue = 0U;
wdog32Config->timeoutValue = 0xFFFFU;
```

See also:

`wdog32_config_t`

Parameters

- config – Pointer to the WDOG32 configuration structure.

`status_t WDOG32_Init(WDOG_Type *base, const wdog32_config_t *config)`

Initializes the WDOG32 module.

This function initializes the WDOG32. To reconfigure the WDOG32 without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
wdog32_config_t config;
WDOG32_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG32_Init(wdog_base,&config);
```

Note: If there is errata ERR010536 (FSL_FEATURE_WDOG_HAS_ERRATA_010536 defined as 1), then after calling this function, user need delay at least 4 LPO clock cycles before accessing other WDOG32 registers.

Parameters

- `base` – WDOG32 peripheral base address.
- `config` – The configuration of the WDOG32.

Return values

- `kStatus_Success` – The initialization was successful
- `kStatus_Timeout` – The initialization timed out

`status_t` WDOG32_Deinit(WDOG_Type *base)

De-initializes the WDOG32 module.

This function shuts down the WDOG32. Ensure that the `WDOG_CS.UPDATE` is 1, which means that the register update is enabled.

Parameters

- `base` – WDOG32 peripheral base address.

Return values

- `kStatus_Success` – The de-initialization was successful
- `kStatus_Timeout` – The de-initialization timed out

`status_t` WDOG32_Unlock(WDOG_Type *base)

Unlocks the WDOG32 register written.

This function unlocks the WDOG32 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- `base` – WDOG32 peripheral base address

Return values

- `kStatus_Success` – The unlock sequence was successful
- `kStatus_Timeout` – The unlock sequence timed out

`void` WDOG32_Enable(WDOG_Type *base)

Enables the WDOG32 module.

This function writes a value into the `WDOG_CS` register to enable the WDOG32. The `WDOG_CS` register is a write-once register. Please check the `enableUpdate` is set to true for calling `WDOG32_Init` to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG32 peripheral base address.

`void` WDOG32_Disable(WDOG_Type *base)

Disables the WDOG32 module.

This function writes a value into the `WDOG_CS` register to disable the WDOG32. The `WDOG_CS` register is a write-once register. Please check the `enableUpdate` is set to true for calling `WDOG32_Init` to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG32 peripheral base address

```
void WDOG32_EnableInterrupts(WDOG_Type *base, uint32_t mask)
```

Enables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to enable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG32 peripheral base address.
- `mask` – The interrupts to enable. The parameter can be a combination of the following source if defined:
 - `kWDOG32_InterruptEnable`

```
void WDOG32_DisableInterrupts(WDOG_Type *base, uint32_t mask)
```

Disables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to disable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG32 peripheral base address.
- `mask` – The interrupts to disabled. The parameter can be a combination of the following source if defined:
 - `kWDOG32_InterruptEnable`

```
static inline uint32_t WDOG32_GetStatusFlags(WDOG_Type *base)
```

Gets the WDOG32 all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;
status = WDOG32_GetStatusFlags(wdog_base) & kWDOG32_RunningFlag;
```

See also:

`_wdog32_status_flags_t`

- `true`: related status flag has been set.
- `false`: related status flag is not set.

Parameters

- `base` – WDOG32 peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

```
void WDOG32_ClearStatusFlags(WDOG_Type *base, uint32_t mask)
```

Clears the WDOG32 flag.

This function clears the WDOG32 status flag.

Example to clear an interrupt flag:

```
WDOG32_ClearStatusFlags(wdog_base,kWDOG32_InterruptFlag);
```

Parameters

- base – WDOG32 peripheral base address.
- mask – The status flags to clear. The parameter can be any combination of the following values:
 - kWDOG32_InterruptFlag

```
void WDOG32_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)
```

Sets the WDOG32 timeout value.

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. To ensure the reconfiguration fits the timing of WCT, unlock function will be called inline.

Parameters

- base – WDOG32 peripheral base address
- timeoutCount – WDOG32 timeout value, count of WDOG32 clock ticks.

```
void WDOG32_SetWindowValue(WDOG_Type *base, uint16_t windowValue)
```

Sets the WDOG32 window value.

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- windowValue – WDOG32 window value.

```
static inline void WDOG32_Refresh(WDOG_Type *base)
```

Refreshes the WDOG32 timer.

This function feeds the WDOG32. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG32 peripheral base address

```
static inline uint16_t WDOG32_GetCounterValue(WDOG_Type *base)
```

Gets the WDOG32 counter value.

This function gets the WDOG32 counter value.

Parameters

- base – WDOG32 peripheral base address.

Returns

Current WDOG32 counter value.

```
WDOG_FIRST_WORD_OF_UNLOCK
```

First word of unlock sequence

```
WDOG_SECOND_WORD_OF_UNLOCK
```

Second word of unlock sequence

WDOG_FIRST_WORD_OF_REFRESH

First word of refresh sequence

WDOG_SECOND_WORD_OF_REFRESH

Second word of refresh sequence

FSL_WDOG32_DRIVER_VERSION

WDOG32 driver version.

enum __wdog32_clock_source

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

Values:

enumerator kWDOG32_ClockSource0

Clock source 0

enumerator kWDOG32_ClockSource1

Clock source 1

enumerator kWDOG32_ClockSource2

Clock source 2

enumerator kWDOG32_ClockSource3

Clock source 3

enum __wdog32_clock_prescaler

Describes the selection of the clock prescaler.

Values:

enumerator kWDOG32_ClockPrescalerDivide1

Divided by 1

enumerator kWDOG32_ClockPrescalerDivide256

Divided by 256

enum __wdog32_test_mode

Describes WDOG32 test mode.

Values:

enumerator kWDOG32_TestModeDisabled

Test Mode disabled

enumerator kWDOG32_UserModeEnabled

User Mode enabled

enumerator kWDOG32_LowByteTest

Test Mode enabled, only low byte is used

enumerator kWDOG32_HighByteTest

Test Mode enabled, only high byte is used

enum `_wdog32_interrupt_enable_t`

WDOG32 interrupt configuration structure.

This structure contains the settings for all of the WDOG32 interrupt configurations.

Values:

enumerator `kWDOG32_InterruptEnable`

Interrupt is generated before forcing a reset

enum `_wdog32_status_flags_t`

WDOG32 status flags.

This structure contains the WDOG32 status flags for use in the WDOG32 functions.

Values:

enumerator `kWDOG32_RunningFlag`

Running flag, set when WDOG32 is enabled

enumerator `kWDOG32_InterruptFlag`

Interrupt flag, set when interrupt occurs

typedef enum `_wdog32_clock_source` `wdog32_clock_source_t`

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

typedef enum `_wdog32_clock_prescaler` `wdog32_clock_prescaler_t`

Describes the selection of the clock prescaler.

typedef struct `_wdog32_work_mode` `wdog32_work_mode_t`

Defines WDOG32 work mode.

typedef enum `_wdog32_test_mode` `wdog32_test_mode_t`

Describes WDOG32 test mode.

typedef struct `_wdog32_config` `wdog32_config_t`

Describes WDOG32 configuration structure.

struct `_wdog32_work_mode`

`#include <fsl_wdog32.h>` Defines WDOG32 work mode.

Public Members

bool `enableWait`

Enables or disables WDOG32 in wait mode

bool `enableStop`

Enables or disables WDOG32 in stop mode

bool `enableDebug`

Enables or disables WDOG32 in debug mode

struct `_wdog32_config`

`#include <fsl_wdog32.h>` Describes WDOG32 configuration structure.

Public Members

`bool enableWdog32`
Enables or disables WDOG32

`wdog32_clock_source_t clockSource`
Clock source select

`wdog32_clock_prescaler_t prescaler`
Clock prescaler value

`wdog32_work_mode_t workMode`
Configures WDOG32 work mode in debug stop and wait mode

`wdog32_test_mode_t testMode`
Configures WDOG32 test mode

`bool enableUpdate`
Update write-once register enable

`bool enableInterrupt`
Enables or disables WDOG32 interrupt

`bool enableWindowMode`
Enables or disables WDOG32 window mode

`uint16_t windowValue`
Window value

`uint16_t timeoutValue`
Timeout value

2.70 CACHE: CACHE Memory Controller

`uint32_t XCACHE_GetInstanceByAddr(uint32_t address)`
brief Returns an instance number given physical memory address.
param address The physical memory address.

Returns

XCACHE instance number starting from 0.

`void XCACHE_EnableCache(XCACHE_Type *base)`
Enables the cache.

Parameters

- base – XCACHE peripheral base address.

`void XCACHE_DisableCache(XCACHE_Type *base)`
Disables the cache.

Parameters

- base – XCACHE peripheral base address.

`void XCACHE_InvalidateCache(XCACHE_Type *base)`
Invalidates the cache.

Parameters

- base – XCACHE peripheral base address.

void XCACHE_InvalidateCacheByRange(uint32_t address, uint32_t size_byte)

Invalidates cache by range.

Note: Address and size should be aligned to “XCACHE_LINESIZE_BYTE”. The startAddr here will be forced to align to XCACHE_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, better to align with cache line size.

void XCACHE_CleanCache(XCACHE_Type *base)

Cleans the cache.

Parameters

- base – XCACHE peripheral base address.

void XCACHE_CleanCacheByRange(uint32_t address, uint32_t size_byte)

Cleans cache by range.

Note: Address and size should be aligned to “XCACHE_LINESIZE_BYTE”. The startAddr here will be forced to align to XCACHE_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, better to align with cache line size.

void XCACHE_CleanInvalidateCache(XCACHE_Type *base)

Cleans and invalidates the cache.

Parameters

- base – XCACHE peripheral base address.

void XCACHE_CleanInvalidateCacheByRange(uint32_t address, uint32_t size_byte)

Cleans and invalidate cache by range.

Note: Address and size should be aligned to “XCACHE_LINESIZE_BYTE”. The startAddr here will be forced to align to XCACHE_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, better to align with cache line size.

```
void XCACHE_EnableWriteBuffer(XCACHE_Type *base, bool enable)
```

Enables/disables the write buffer.

Parameters

- base – XCACHE peripheral base address.
- enable – The enable or disable flag. true - enable the write buffer. false - disable the write buffer.

```
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 XCACHE_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_XCACHE_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, better to align with cache line size.

```
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 XCACHE_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_XCACHE_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, better to align with cache line size.

```
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 XCACHE_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_XCACHE_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, better to align with cache line size.

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 XCACHE_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_XCACHE_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, better to align with cache line size.

FSL_CACHE_DRIVER_VERSION

cache driver version.

XCACHE_LINESIZE_BYTE

cache line size.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called “packet-driven BDM” interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT](<https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/>) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to [FreeMASTER community](#) or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a “middleware” component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pd_bdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER “middleware” driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample applications Zephyr sample applications demonstrate Kconfig and Device Tree configuration which configure the FreeMASTER middleware module for a selected communication option (Serial, CAN, Network or RTT).

Refer to *readme.md* files in each sample directory for description of configuration options required to implement FreeMASTER connectivity.

Description

This section shows how to add the FreeMASTER Communication Driver into application and how to configure the connection to the FreeMASTER visualization tool.

Features The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space.
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables.
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control an embedded application.

Board Detection The FreeMASTER protocol V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

Memory Read This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must be shorter than the MTU to fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The driver uses size-aligned operations to read the target memory (for example; uses proper read-word instruction when an address is aligned to 4 bytes).

Memory Write Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; uses proper write-word instruction when an address is aligned to 4 bytes).

Masked Memory Write To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

Oscilloscope The protocol and driver enables any number of variables to be read at once with a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

The driver can be configured to support any number of Oscilloscope instances and enable simultaneously running graphs to be displayed on the host computer screen.

Recorder The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

The driver can be configured to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables setting the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

TSA With the TSA feature, data types and variables can be described directly in the application source code. Such information is later provided to the FreeMASTER tool which may use it instead of reading symbol data from the application ELF executable file.

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

TSA Safety When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

Application commands The Application Commands are high-level messages that can be delivered from the PC Host to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing in general.

Pipes The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losses.

Serial single-wire operation The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- “External” single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- “True” single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as “automatic” in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the .c files must be added to the project, compiled, and linked together with the application.
 - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
 - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
 - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
 - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
 - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
 - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.
- **src/drivers/[sdk]/serial** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- **src/drivers/[sdk]/can** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- **src/drivers/[sdk]/network** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
 - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
 - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR    [0|1]
#define FMSTR_SHORT_INTR  [0|1]
#define FMSTR_POLL_DRIVEN [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See [Driver interrupt modes](#).

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- FMSTR_SERIAL - serial communication protocol
- FMSTR_CAN - using CAN communication
- FMSTR_PDBDM - using packet-driven BDM communication
- FMSTR_NET - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as **FMSTR_SERIAL_DRV**. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32...255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0...255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the “True” single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE


```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetCanBaseAddress()` to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options

FMSTR_DISABLE

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+@W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options**FMSTR_USE_SCOPE**

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options**FMSTR_USE_RECORDER**

```
#define FMSTR_USE_RECORDER [number]
```

Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.
Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature. Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using FMSTR_RegisterAppCmdCall(). Default value is 0.

TSA options**FMSTR_USE_TSA**

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool. Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables. Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project. Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions. Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used. Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support. The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_QUEUE_SIZE*) and the character time is the time needed to transmit or receive a single byte over the SCI line.

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

In this mode, both the communication and the FreeMASTER protocol decoding are done in the *FMSTR_Poll* routine. No interrupts are needed and the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, and similar handlers compile to an empty code.

When using this mode, ensure that the *FMSTR_Poll* function is called by the application at least once per the serial "character time" which is the time needed to transmit or receive a single character.

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the *FMSTR_Poll* routine. An application interrupt can occur in the middle of the

Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

The same issue may appear even in the full interrupt mode (FMSTR_LONG_INTR), if volatile variables are modified in the interrupt code with a priority higher than the priority of the communication interrupt.

Data types Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the platform-specific header file.

To prevent name conflicts with the symbols used in the application, all data types, macros, and functions have the FMSTR_ prefix. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

Communication interface initialization The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is the application startup code responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

Note: It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

FreeMASTER Recorder calls When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere else. The example applications provided together with the driver code call the FMSTR_Recorder in the main application loop.

In applications where FMSTR_Recorder is called periodically with a constant period, specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

Driver usage Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all *.c files of the FreeMASTER driver from the `src/common/platforms/[your_platform]` folder are a part of the project. See [Driver files](#) for more details.
- Configure the FreeMASTER driver by creating or editing the `freemaster_cfg.h` file and by saving it into the application project directory. See [Driver configuration](#) for more details.
- Include the `freemaster.h` file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, call the FMSTR_Poll API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally so that the interrupts can be handled by the CPU.

Communication troubleshooting The most common problem that causes communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When a communication between the PC Host running FreeMASTER and the target MCU cannot be established, try enabling the FMSTR_DEBUG_TX option in the `freemaster_cfg.h` file and call the FMSTR_Poll function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Driver API

This section describes the driver Application Programmers' Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

Control API There are three key functions to initialize and use the driver.

FMSTR_Init

Prototype

```
FMSTR_BOOL FMSTR_Init(void);
```

- Declaration: `freemaster.h`
- Implementation: `freemaster_protocol.c`

Description This function initializes the internal variables of the FreeMASTER driver and enables the communication interface. This function does not change the configuration of the selected communication module. The hardware module must be initialized before the *FMSTR_Init* function is called.

A call to this function must occur before calling any other FreeMASTER driver API functions.

FMSTR_Poll

Prototype

```
void FMSTR_Poll(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see *Driver interrupt modes*). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the *FMSTR_Poll* function is called during the “idle” time in the main application task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the *FMSTR_Poll* function is called at least once per the time calculated as:

$$N * Tchar$$

where:

- *N* is equal to the length of the receive FIFO queue (configured by the *FMSTR_COMM_RQUEUE_SIZE* macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

Note: In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void);
void FMSTR_CanIsr(void);
```

- Declaration: *freemaster.h*
- Implementation: *hw-specific low-level driver C file*

Description This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see *Driver interrupt modes*), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

Note: In a poll-driven mode, this function is compiled as an empty function and does not have to be used.

Recorder API

FMSTR_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance 0 which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the *FMSTR_REC_BUFF_SIZE* and *FMSTR_REC_TIMEBASE* options.

For more information, see [Configurable items](#).

FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance *recIndex*. If the selected Recorder is not active when the *FMSTR_Recorder* function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down the follow-up samples (number of *FMSTR_Recorder* function calls) and de-activates the Recorder when the required post-trigger samples are finished.

The *FMSTR_Recorder* function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

FMSTR_RecorderTrigger

Prototype

```
void FMSTR_RecorderTrigger(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application code for programmatic control over the Recorder triggering. This can be useful when a more complex triggering conditions need to be used.

Fast Recorder API The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

TSA Tables When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), it defines the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always exactly one TSA Table List defined, which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. The symbols can then be used to create FreeMASTER variables for visualization or control.

TSA table definition The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide an access to the individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-language symbol.

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

After this opening macro, the TSA descriptors are placed using these macros:

```
/* Adding variable descriptors */
FMSTR_TSA_RW_VAR(name, type) /* read/write variable entry */
FMSTR_TSA_RO_VAR(name, type) /* read-only variable entry */

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
FMSTR_TSA_MEMBER(struct_name, member_name, type) /* structure member entry */

/* Memory blocks */
FMSTR_TSA_RW_MEM(name, type, address, size) /* read/write memory block */
FMSTR_TSA_RO_MEM(name, type, address, size) /* read-only memory block */
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

TSA descriptor parameters The TSA descriptor macros accept these parameters:

- *name* — variable name. The variable must be defined before the TSA descriptor references it.
- *type* — variable or member type. Only one of the pre-defined type constants may be used (see below).
- *struct_name* — structure type name. The type must be defined (typedef) before the TSA descriptor references it.

- *member_name* — structure member name.

Note: The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table.

Note: To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

TSA variable types The table lists *type* identifiers which can be used in TSA descriptors:

Constant	Description
FMSTR_TSA_UINTn	Unsigned integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size <i>n</i> bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(<i>m,n</i>)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(<i>m,n</i>)	Unsigned fractional number in general UQ form (m+n total bits)
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type
FMSTR_TSA_POINTER	Generic pointer type defined (platform-specific 16 or 32 bit)
FM-STR_TSA_USERTYPE(<i>name</i>)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()
```

```
FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *fmrstr:* protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)
```

```
/* Directory entry applies to all subsequent MEMFILE entries */
```

```
FMSTR_TSA_DIRECTORY("/text_files") /* entering a new virtual directory */
```

(continues on next page)

(continued from previous page)

```

/* The readme.txt file will be accessible at the fmstr://text_files/readme.txt URL */
FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt)) /* memory-mapped file */

/* Files can also be specified with a full path so the DIRECTORY entry does not apply */
FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index)) /* memory-mapped file */
FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) /* memory-mapped file */

/* Hyperlinks can point to a local MEMFILE object or to the Internet */
FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

/* Project file links simplify opening the projects from any URLs */
FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
FMSTR_TSA_PROJECT("Full Project (online)", "http://mycompany.com/prj/demo.pmp")

FMSTR_TSA_TABLE_END()

```

TSA API

FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR buffAddr, FMSTR_SIZE buffSize);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *buffAddr* [in] - address of the memory buffer for the dynamic TSA table
- *buffSize* [in] - size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

FMSTR_TsaAddVar

Prototype

```

FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR tsaName, FMSTR_TSATBL_STRPTR
↪ tsaType,
    FMSTR_TSATBL_VOIDPTR varAddr, FMSTR_SIZE32 varSize,
    FMSTR_SIZE flags);

```

- Declaration: *freemaster.h*

- Implementation: *freemaster_tsa.c*

Arguments

- *tsaName* [in] - name of the object
- *tsaType* [in] - name of the object type
- *varAddr* [in] - address of the object
- *varSize* [in] - size of the object
- *flags* [in] - access flags; a combination of these values:
 - *FMSTR_TSA_INFO_RO_VAR* — read-only memory-mapped object (typically a variable)
 - *FMSTR_TSA_INFO_RW_VAR* — read/write memory-mapped object
 - *FMSTR_TSA_INFO_NON_VAR* — other entry, describing structure types, structure members, enumerations, and other types

Description This function can be called only when the dynamic TSA table is enabled by the *FMSTR_USE_TSA_DYNAMIC* configuration option and when the *FMSTR_SetUpTsaBuff* function call is made to assign the dynamic TSA table memory. This function adds an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See [TSA table definition](#) for more details about the TSA table entries.

Application Commands API

FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Description This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the *FMSTR_APPCMDRESULT_NOCMD* constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the *FMSTR_AppCmdAck* call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The *FMSTR_GetAppCmd* function does not report the commands for which a callback handler function exists. If the *FMSTR_GetAppCmd* function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns *FMSTR_APPCMDRESULT_NOCMD*.

FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* dataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *dataLen* [out] - pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see [FMSTR_GetAppCmd](#)).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the FMSTR_AppCmdAck call, copy the data out to a private buffer.

FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT resultCode);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultCode* [in] - the result code which is to be returned to FreeMASTER

Description This function is used when the Application Command processing finishes in the application. The resultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(FMSTR_ADDR resultDataAddr, FMSTR_SIZE resultDataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultDataAddr* [in] - pointer to the data buffer that is to be copied to the Application Command data buffer
- *resultDataLen* [in] - length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description This function can be used before the Application Command processing finishes, when there are data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use FMSTR_GetAppCmdData and the data buffer after FMSTR_AppCmdSetResponseData is called.

Note: The current version of FreeMASTER does not support the Application Command response data.

FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(FMSTR_APPCMD_CODE appCmdCode, FMSTR_
↪PAPPCMDFUNC callbackFunc);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *appCmdCode* [in] - the Application Command code for which the callback is to be registered
- *callbackFunc* [in] - pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return value This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when trying to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is

```
FMSTR_APPCMD_RESULT HandlerFunction(FMSTR_APPCMD_CODE nAppcmd,
FMSTR_APPCMD_PDATA pData, FMSTR_SIZE nDataLen);
```

Where:

- *nAppcmd* -Application Command code
- *pData* —points to the Application Command data received (if any)
- *nDataLen* —information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

Note: The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the FMSTR_RegisterAppCmdCall function always fails.

Pipes API

FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT pipePort, FMSTR_PPIPEFUNC pipeCallback,
    FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize,
    FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize,
    FMSTR_U8 type, const FMSTR_CHAR *name);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipePort* [in] - port number that identifies the pipe for the client
- *pipeCallback* [in] - pointer to the callback function that is called whenever a pipe data status changes
- *pipeRxBuff* [in] - address of the receive memory buffer
- *pipeRxSize* [in] - size of the receive memory buffer
- *pipeTxBuff* [in] - address of the transmit memory buffer
- *pipeTxSize* [in] - size of the transmit memory buffer
- *type* [in] - a combination of FMSTR_PIPE_MODE_XXX and FMSTR_PIPE_SIZE_XXX constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.
- *name* [in] - user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

Description This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the FMSTR_PipeRead call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the FMSTR_PipeWrite call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The *PipeHandler* name is only a placeholder and must be defined by the application.

```
void PipeHandler(FMSTR_HPIPE pipeHandle);
```

FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call

Description This function de-initializes the pipe object. No data can be received or sent on the pipe after this call.

FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,  
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data to be written
- *pipeDataLen* [in] - length of the data to be written
- *writeGranularity* [in] - size of the minimum unit of data which is to be written

Description This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the *nGranularity* value equal to the *nLength* value, all data are considered as one chunk which is either written successfully as a whole or not at all. The *nGranularity* value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE readGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data buffer to be filled with the received data
- *pipeDataLen* [in] - length of the data to be read
- *readGranularity* [in] - size of the minimum unit of data which is to be read

Description This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The readGranularity argument can be used to copy the data in larger chunks in the same way as described in the FMSTR_PipeWrite function.

API data types This section describes the data types used in the FreeMASTER driver. The information provided here can be useful when modifying or porting the FreeMASTER Communication Driver to new NXP platforms.

Note: The licensing conditions prohibit use of FreeMASTER and the FreeMASTER Communication Driver with non-NXP MPU or MCU products.

Public common types The table below describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Type name	Description
<i>FM-STR_ADDR</i> For example, this type is defined as long integer on the 56F8xxx platform where the 24-bit addresses must be supported, but the C-pointer may be only 16 bits wide in some compiler configurations.	Data type used to hold the memory address. On most platforms, this is normally a C-pointer, but it may also be a pure integer type.
<i>FM-STR_SIZE</i> It is required that this type is unsigned and at least 16 bits wide integer.	Data type used to hold the memory block size.
<i>FM-STR_BOOL</i> This type is used only in zero/non-zero conditions in the driver code.	Data type used as a general boolean type.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command code.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to create the Application Command data buffer.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command result code.

Public TSA types The table describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included in the user application indirectly by the *freemaster.h* file.

<i>FM-STR_TSA_TII</i>	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables. By default, this is defined as <i>FM-STR_SIZE</i> .
<i>FM-STR_TSA_TS</i>	Data type used to hold a memory block size, as used in the TSA descriptors. By default, this is defined as <i>FM-STR_SIZE</i> .

Public Pipes types The table describes the data types used by the FreeMASTER Pipes API:

<i>FM-STR_HPIPE</i>	Pipe handle that identifies the open-pipe object. Generally, this is a pointer to a void type.
<i>FM-STR_PIPE_PC</i>	Integer type required to hold at least 7 bits of data. Generally, this is an unsigned 8-bit or 16-bit type.
<i>FM-STR_PIPE_SI</i>	Integer type required to hold at least 16 bits of data. This is used to store the data buffer sizes.
<i>FM-STR_PPIPEF</i>	Pointer to the pipe handler function. See FM-STR_PipeOpen for more details.

Internal types The table describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and they are not available in the application code.

<i>FMSTR_U8</i>	The smallest memory entity.
On the vast majority of platforms, this is an unsigned 8-bit integer.	
On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.	
<i>FMSTR_U16</i>	Unsigned 16-bit integer.
<i>FMSTR_U32</i>	Unsigned 32-bit integer.
<i>FMSTR_S8</i>	Signed 8-bit integer.
<i>FMSTR_S16</i>	Signed 16-bit integer.
<i>FMSTR_S32</i>	Signed 32-bit integer.
<i>FMSTR_FLOAT</i>	4-byte standard IEEE floating-point type.
<i>FMSTR_FLAGS</i>	Data type forming a union with a structure of flag bit-fields.
<i>FMSTR_SIZE8</i>	Data type holding a general size value, at least 8 bits wide.
<i>FMSTR_INDEX</i>	General for-loop index. Must be signed, at least 16 bits wide.
<i>FMSTR_BCHR</i>	A single character in the communication buffer.
Typically, this is an 8-bit unsigned integer, except for the DSP platforms where it is a 16-bit integer.	
<i>FMSTR_BPTR</i>	A pointer to the communication buffer (an array of <i>FMSTR_BCHR</i>).

Document references

Links

- This document online: <https://mcuxpresso.nxp.com/mcuxsdk/latest/html/middleware/freemaster/doc/index.html>

- FreeMASTER tool home: www.nxp.com/freemaster
- FreeMASTER community area: community.nxp.com/community/freemaster
- FreeMASTER GitHub code repo: <https://github.com/nxp-mcuxpresso/mcux-freemaster>
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

- *FreeMASTER Usage Serial Driver Implementation* (document [AN4752](#))
- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document [AN4771](#))
- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document [AN4860](#))

Revision history This Table summarizes the changes done to this document since the initial release.

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support.
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. added MCAN interface. Folders structure at the installation destination was rearranged.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.
4.3	10/2024	Added description of Network and Segger J-Link RTT interface configuration. Accompanying the MCUXpresso SDK version 24.12.00.
4.4	04/2025	Added Zephyr-specific information. Accompanying the MCUXpresso SDK version 25.06.00.

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

[Readme](#)

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme