



MCUXpresso SDK Documentation

Release 25.06.00



NXP
Jun 26, 2025



Table of contents

1	LPCXpresso51U68	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	31
1.3.1	Getting Started with MCUXpresso SDK Repository	31
1.4	Release Notes	44
1.4.1	MCUXpresso SDK Release Notes	44
1.5	ChangeLog	48
1.5.1	MCUXpresso SDK Changelog	48
1.6	Driver API Reference Manual	84
1.7	Middleware Documentation	84
1.7.1	FreeMASTER	85
1.7.2	FreeRTOS	85
2	LPC51U68	87
2.1	Clock Driver	87
2.2	CRC: Cyclic Redundancy Check Driver	107
2.3	CTIMER: Standard counter/timers	110
2.4	DMA: Direct Memory Access Controller Driver	119
2.5	FLASHIAP: Flash In Application Programming Driver	136
2.6	FLEXCOMM: FLEXCOMM Driver	141
2.7	FLEXCOMM Driver	141
2.8	FMEAS: Frequency Measure Driver	142
2.9	GINT: Group GPIO Input Interrupt Driver	143
2.10	I2C: Inter-Integrated Circuit Driver	146
2.11	I2C DMA Driver	146
2.12	I2C Driver	148
2.13	I2C Master Driver	151
2.14	I2C Slave Driver	161
2.15	I2S: I2S Driver	170
2.16	I2S DMA Driver	170
2.17	I2S Driver	173
2.18	IAP: In Application Programming Driver	182
2.19	INPUTMUX: Input Multiplexing Driver	188
2.20	Common Driver	193
2.21	ADC: 12-bit SAR Analog-to-Digital Converter Driver	205
2.22	GPIO: General Purpose I/O	216
2.23	IOCON: I/O pin configuration	219
2.24	MRT: Multi-Rate Timer	220
2.25	PINT: Pin Interrupt and Pattern Match Driver	224
2.26	Power Driver	232
2.27	Reset Driver	237
2.28	RTC: Real Time Clock	240
2.29	SCTimer: SCTimer/PWM (SCT)	245
2.30	SPI: Serial Peripheral Interface Driver	262

2.31	SPI DMA Driver	262
2.32	SPI Driver	266
2.33	USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver	274
2.34	USART DMA Driver	274
2.35	USART Driver	277
2.36	UTICK: MicroTick Timer Driver	293
2.37	WWDG: Windowed Watchdog Timer Driver	294
3	Middleware	299
3.1	Motor Control	299
3.1.1	FreeMASTER	299
4	RTOS	337
4.1	FreeRTOS	337
4.1.1	FreeRTOS kernel	337
4.1.2	FreeRTOS drivers	337
4.1.3	backoffalgorithm	337
4.1.4	corehttp	337
4.1.5	corejson	337
4.1.6	coremqtt	338
4.1.7	coremqtt-agent	338
4.1.8	corepkcs11	338
4.1.9	freertos-plus-tcp	338

This documentation contains information specific to the lpcxpresso51u68 board.

Chapter 1

LPCXpresso51U68

1.1 Overview

The LPCXpresso family of boards provides a powerful and flexible development system for NXP's Cortex-M MCUs. The LPCXpresso51U68 board has been developed by NXP to enable evaluation of and prototyping with the LPC51U68 family of MCUs and its low power features make it as easy as possible to get started with your project. LPCXpresso is a low-cost development platform available from NXP supporting NXP's ARM-based microcontrollers. The platform is comprised of a simplified Eclipse-based IDE and low-cost target boards which include an attached JTAG debugger. LPCXpresso is an end-to-end solution enabling embedded engineers to develop their applications from initial evaluation to final production.



MCU device and part on board is shown below:

- Device: LPC51U68
- PartNumber: LPC51U68JBD64

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

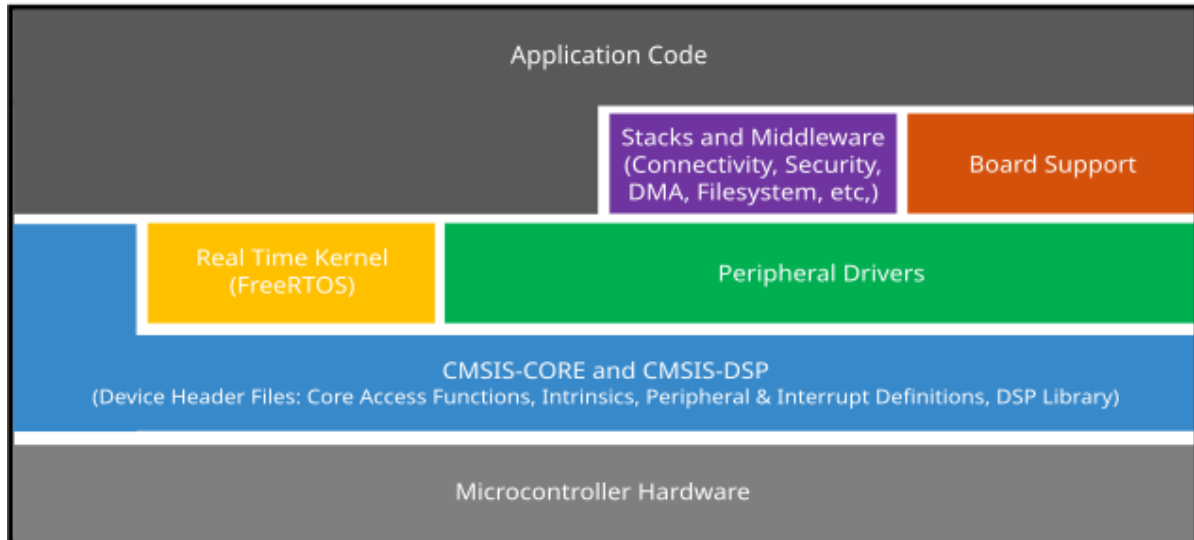
Overview

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

such as FreeRTOS and Azure RTOS, a USB host and device stack, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for LPCXpresso51U68* (document MCUXSDKLPC51U68RN).

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



MCUXpresso SDK board support package folders

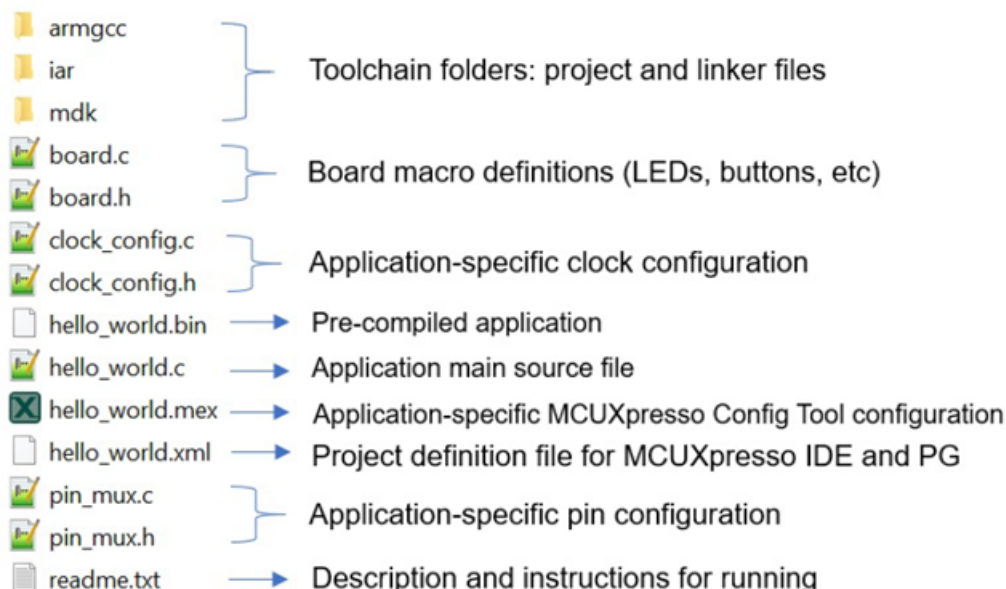
MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm® Cortex®-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top level boards folder and each supported board has its own folder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

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

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

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

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

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

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

Run a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the LPCXpresso51U68 hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

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

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

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

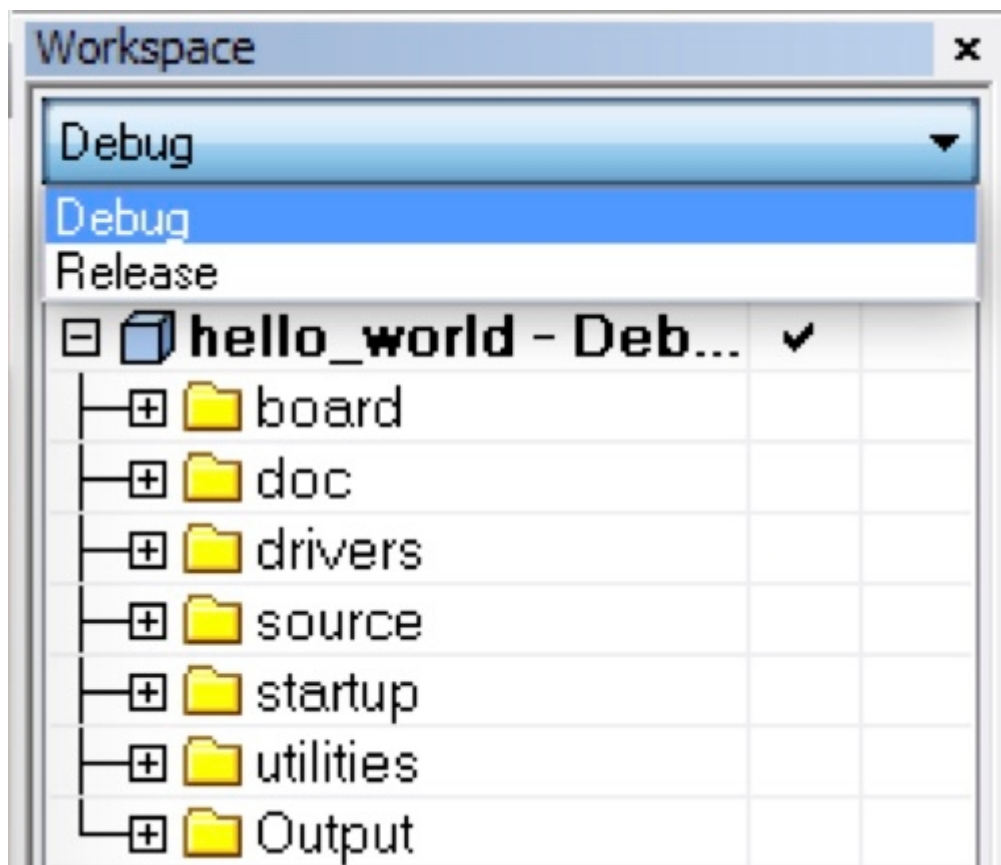
Using the LPCXpresso51U68 hardware platform as an example, the `hello_world` workspace is located in:

```
<install_dir>/boards/lpcxpresso51U68/demo_apps/hello_world/iar/hello_world.eww
```

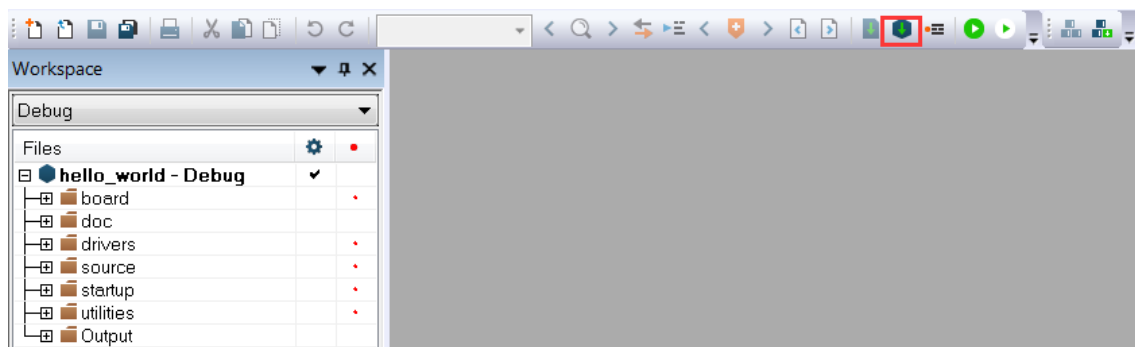
Other example applications may have additional folders in their path.

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.

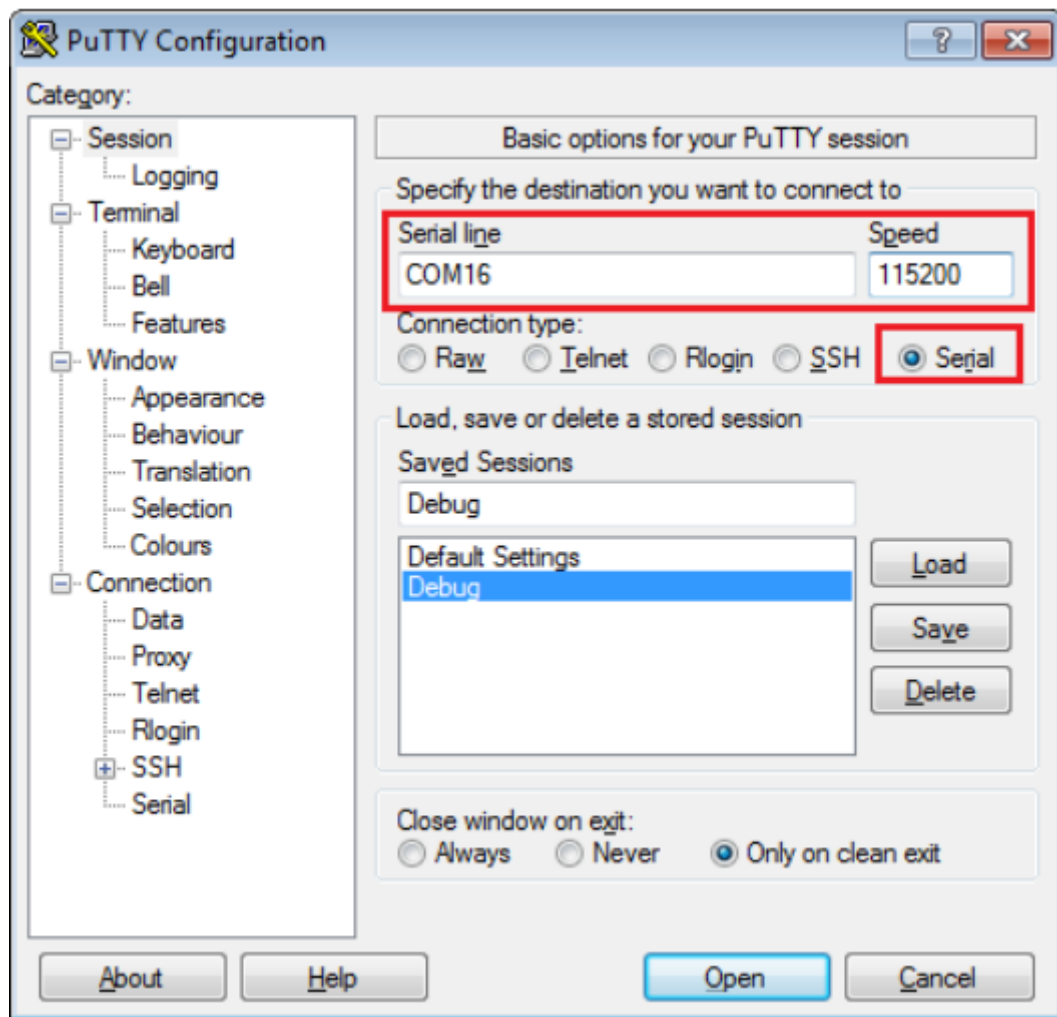


4. The build completes without errors.

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

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

1. Download and install LPCScript or the Windows® operating systems driver for LPCXpresso boards from www.nxp.com/lpcutilities. This installs the required drivers for the board.
2. Connect the development platform to your PC via USB cable between the Link2 USB connector (named Link for some boards) and the PC USB connector. If you are connecting for the first time, allow about 30 seconds for the devices to enumerate.
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 Appendix A). Configure the terminal with these settings:
 1. 115200 baud rate (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
 2. No parity
 3. 8 data bits
 4. 1 stop bit

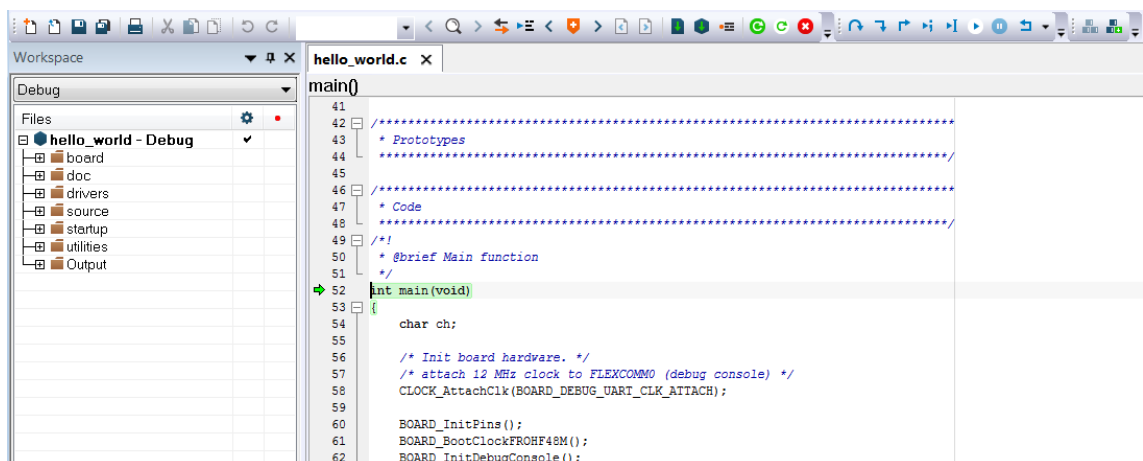


4. In IAR, click the “Download and Debug” button to download the application to the target.



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

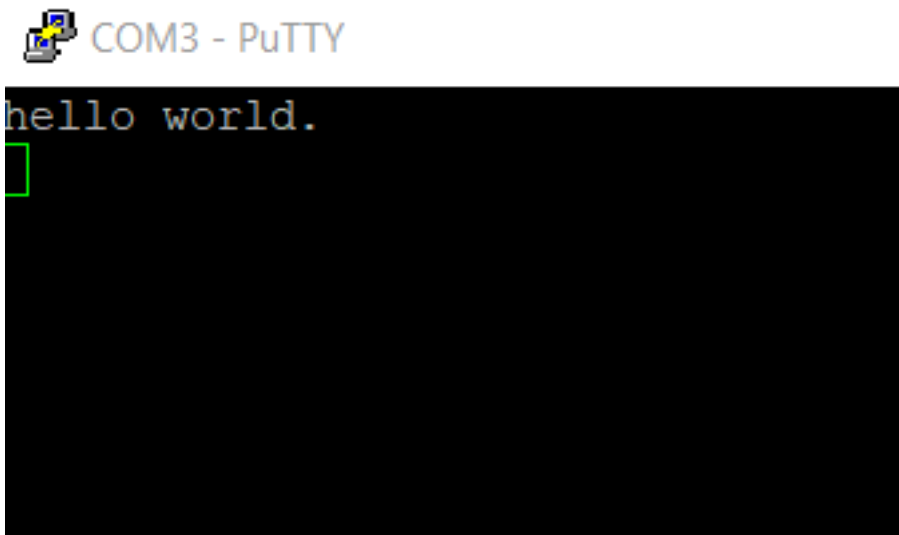
Note: The application is programmed to the external on board flash, then jumped to SRAM to run



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



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



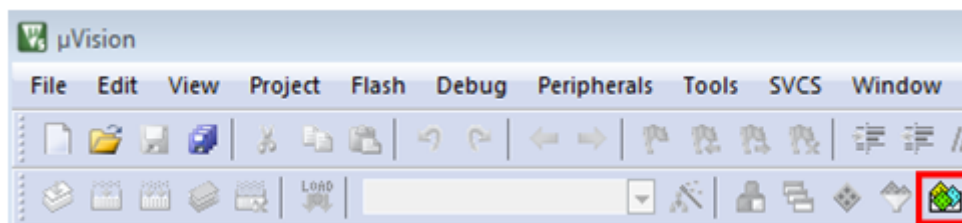
Parent topic: [Run a demo application using IAR](#)

Run a demo using Keil® MDK/μVision

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The hello_world demo application targeted for the LPCXpresso51U68 hardware platform is used as an example, although these steps can be applied to any demo or example application in the MCUXpresso SDK.

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

1. Open the MDK IDE, which is called μVision. In the IDE, select the **Pack Installer** icon.



2. After the installation finishes, close the Pack Installer window and return to the μVision IDE.

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

Build an example application

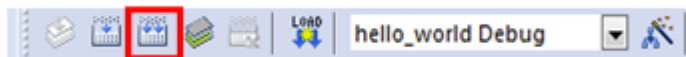
1. Open the desired example application workspace in:

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

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

```
<install_dir>/boards/lpcxpresso51U68/demo_apps/hello_world/mdk/hello_world.uvmpw
```

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

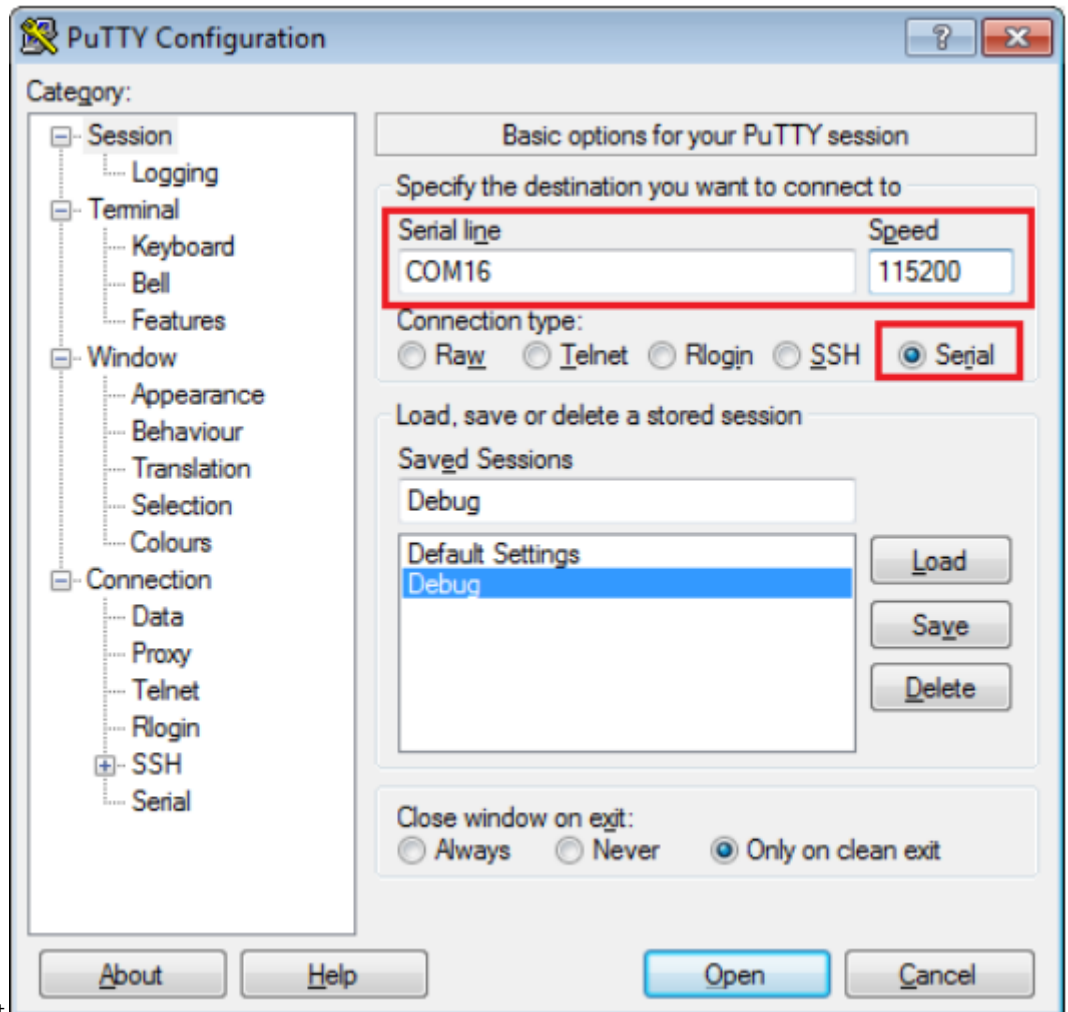


3. The build completes without errors.

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

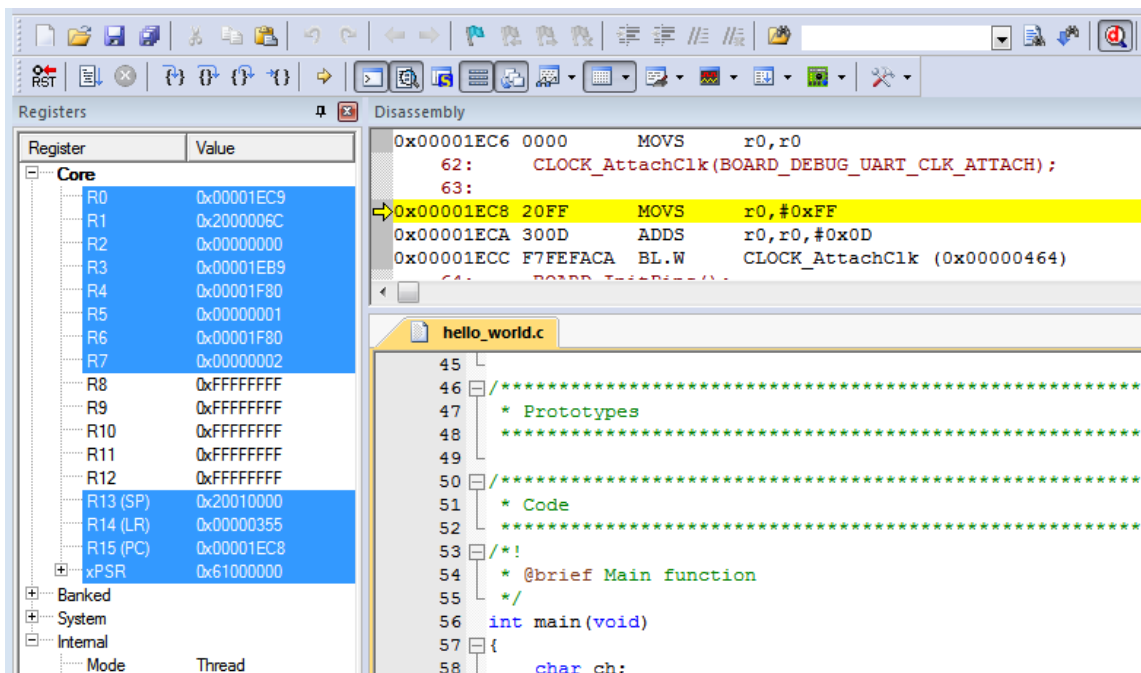
Run an example application To download and run the application, perform these steps:

1. Download and install LPCScript or the Windows® operating systems driver for LPCXpresso boards from www.nxp.com/lpcutilities. This installs the required drivers for the board.
2. Connect the development platform to your PC via USB cable between the Link2 USB connector and the PC USB connector. If you are connecting for the first time, allow about 30 seconds for the devices to enumerate.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see Appendix A). Configure the terminal with these settings:
 1. 115200 baud rate (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
 2. No parity
 3. 8 data bits

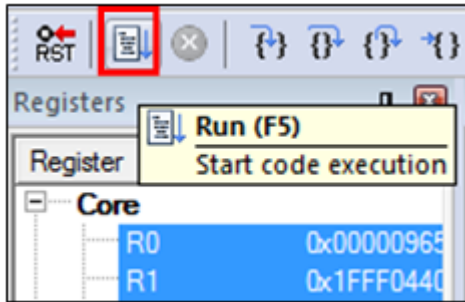


4. 1 stop bit

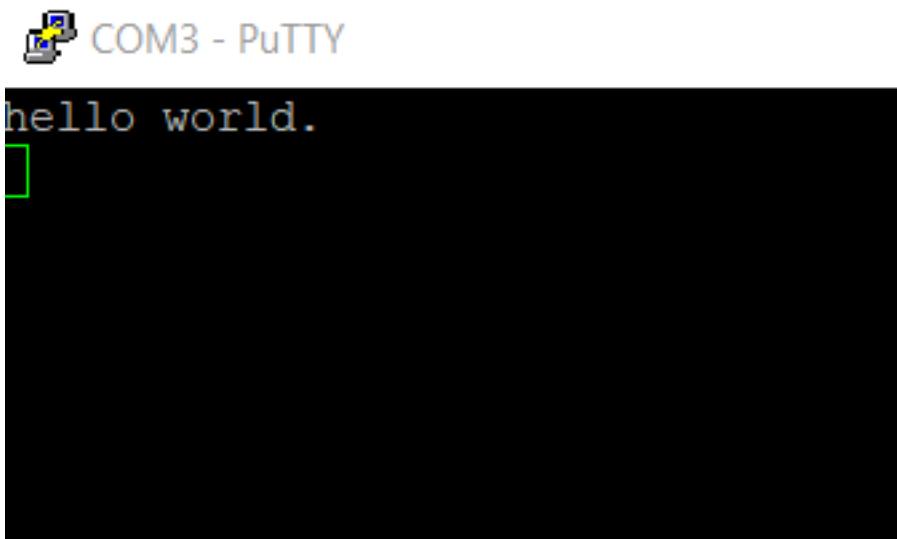
4. To debug the application, click the “Start/Stop Debug Session” button, highlighted in red.



5. Run the code by clicking the “Run” button to start the application.



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



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

Run a demo using Arm® GCC

This section describes the steps to configure the command line Arm® GCC tools to build, run, and debug demo applications and necessary driver libraries provided in the MCUXpresso SDK. The `hello_world` demo application is targeted for the LPCXpresso51U68 hardware platform which is used as an example.

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

Install GCC Arm Embedded tool chain Download and run the installer from launchpad.net/gcc-arm-embedded. This is the actual toolset (in other words, compiler, linker, etc.). The GCC toolchain should correspond to the latest supported version, as described in the *MCUXpresso SDK Release Notes Supporting LPCXpresso51U68*. (document MCUXSD-KLPC51U68RN).

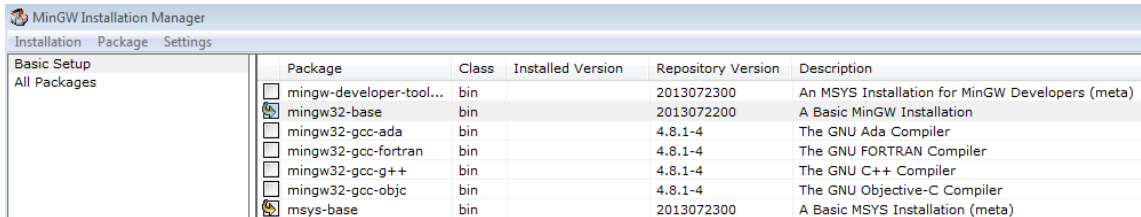
Parent topic: Set up toolchain

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

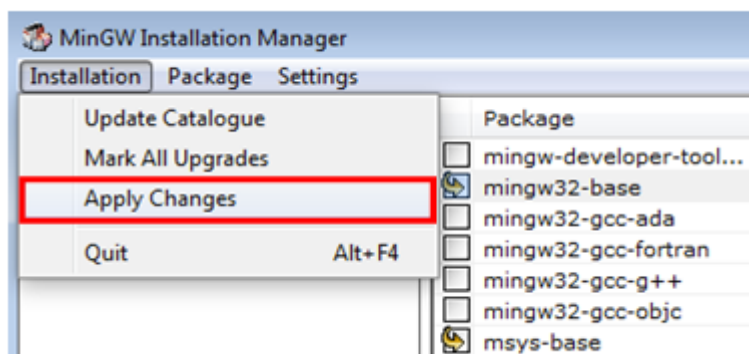
1. Download the latest MinGW mingw-get-setup installer from [MinGW](#).
2. Run the installer. The recommended installation path is C:\MinGW, however, you may install to any location.

Note: The installation path cannot contain any spaces.

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



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

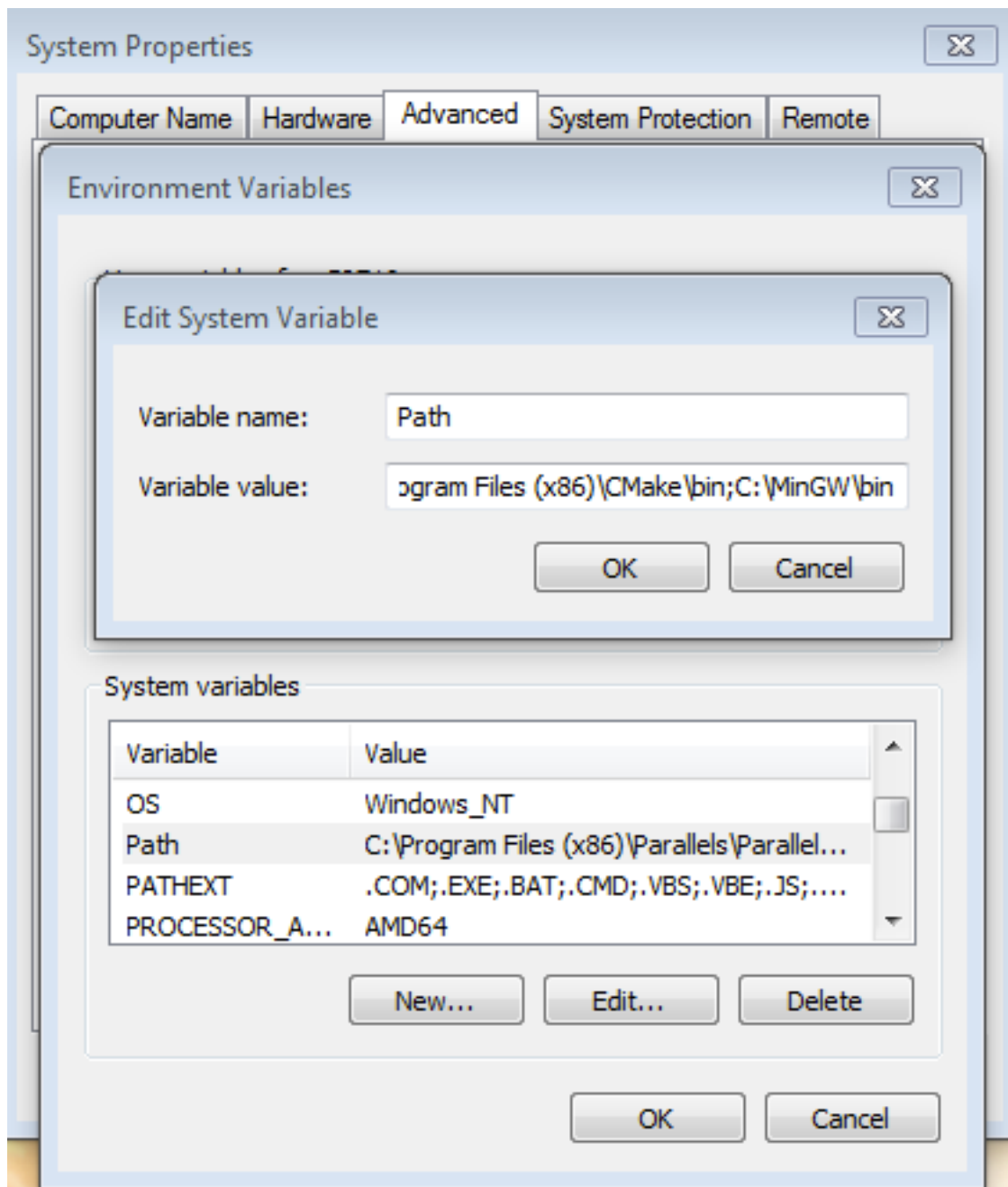


5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel->System and Security->System->Advanced System Settings** in the **Environment Variables...** section. The path is:

```
<mingw_install_dir>\bin
```

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

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



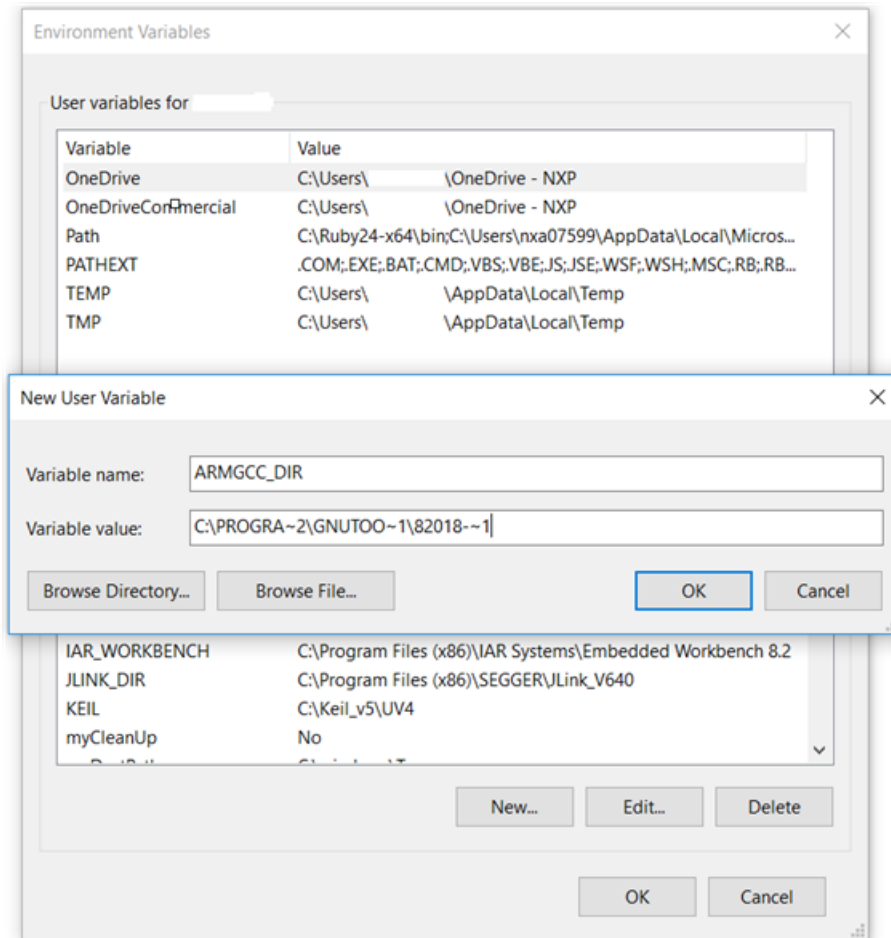
Parent topic:Set up toolchain

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

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

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

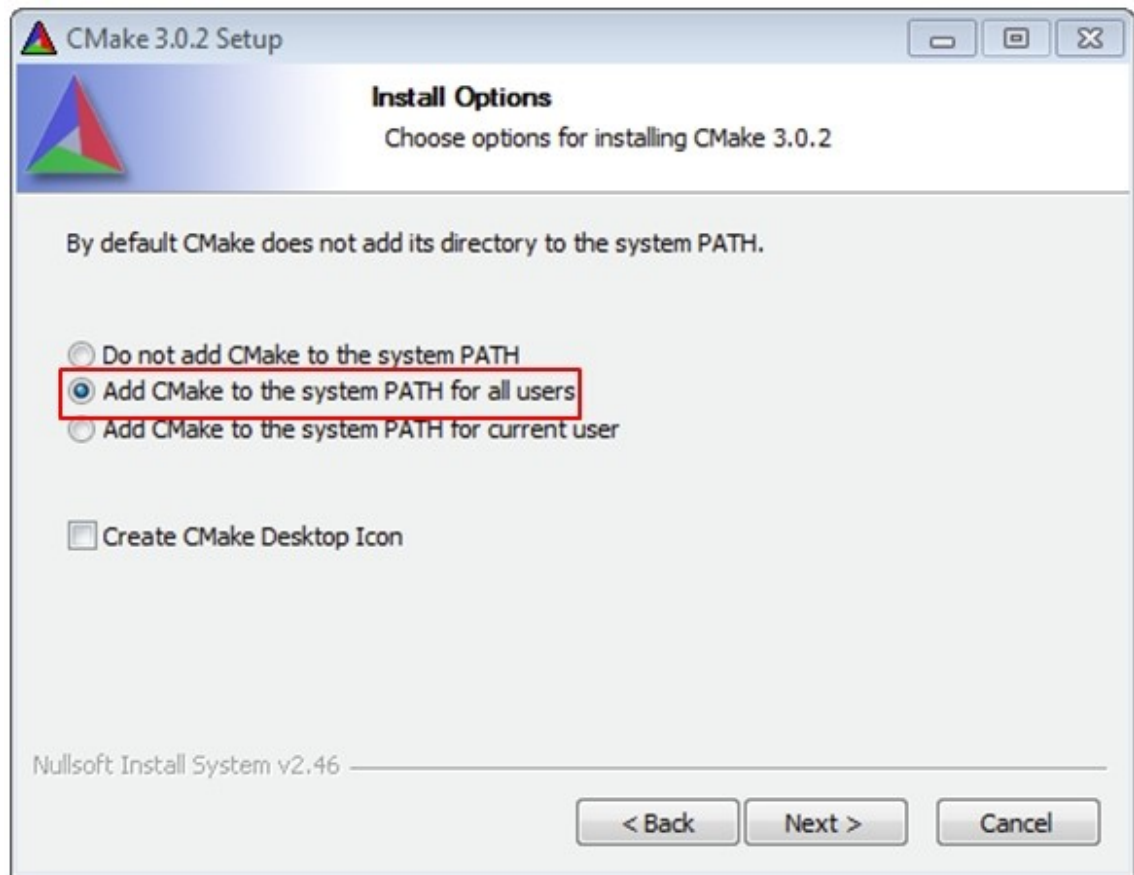
```
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>for %I in (.) do echo %~sI
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>echo C:\PROGRA~2\GNUTOO~1\82018~1
C:\PROGRA~2\GNUTOO~1\82018~1
```



Parent topic: Set up toolchain

Install CMake

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



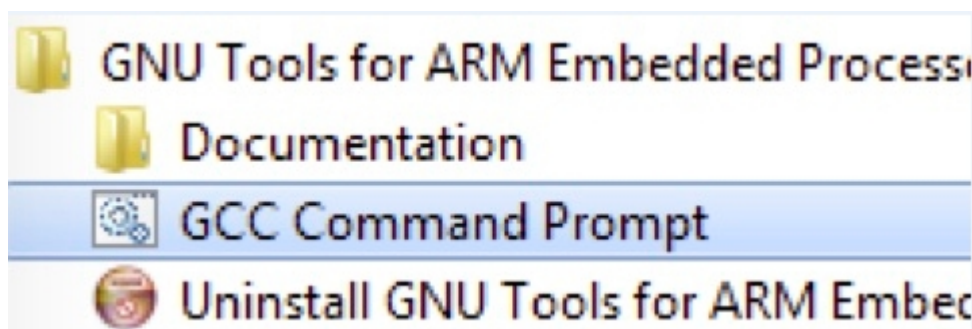
3. Follow the remaining instructions of the installer.
4. You may need to reboot your system for the PATH changes to take effect.
5. Make sure `sh.exe` is not in the Environment Variable PATH. This is a limitation of mingw32-make.

Parent topic:Set up toolchain

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

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

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



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

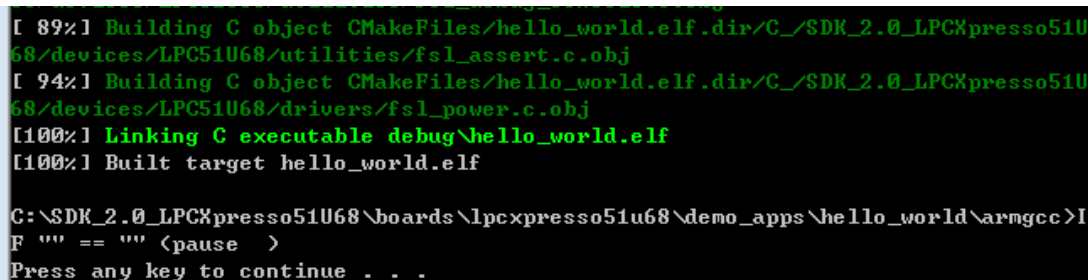
```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc
```

For this example, the exact path is:

```
<install_dir>/examples/lpcxpresso51U68/demo_apps/hello_world/armgcc
```

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

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



```
[ 89%] Building C object CMakeFiles/hello_world.elf.dir/C:/SDK_2.0_LPCXpresso51U68/devices/LPC51U68/utilities/fsl_assert.c.obj
[ 94%] Building C object CMakeFiles/hello_world.elf.dir/C:/SDK_2.0_LPCXpresso51U68/devices/LPC51U68/drivers/fsl_power.c.obj
[100%] Linking C executable debug\hello_world.elf
[100%] Built target hello_world.elf

C:\SDK_2.0_LPCXpresso51U68\boards\lpcxpresso51u68\demo_apps\hello_world\armgcc>I
F "" == "" <pause >
Press any key to continue . . .
```

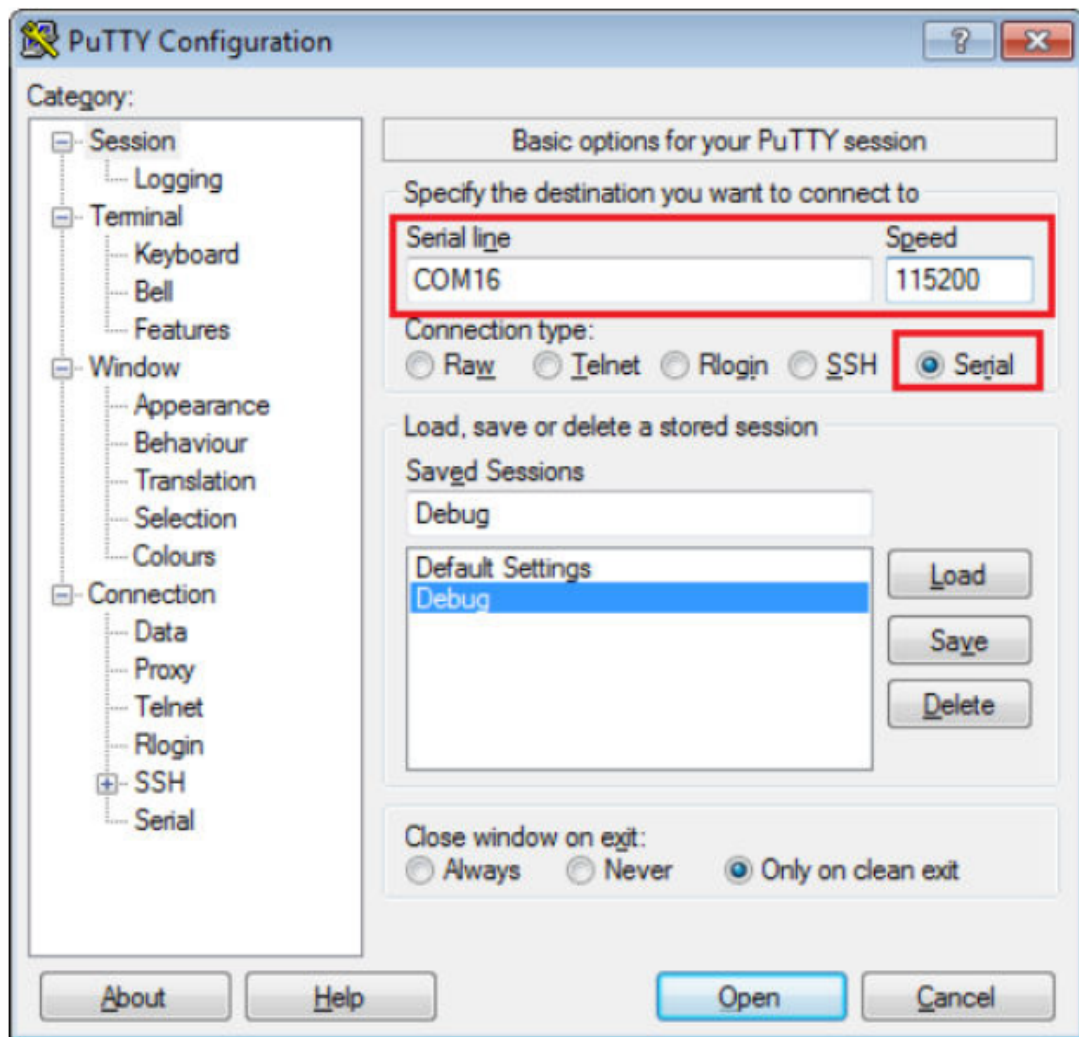
Parent topic: [Run a demo using Arm® GCC](#)

Run an example application This section describes steps to run a demo application using J-Link GDB Server application. To perform this exercise, two things must be done:

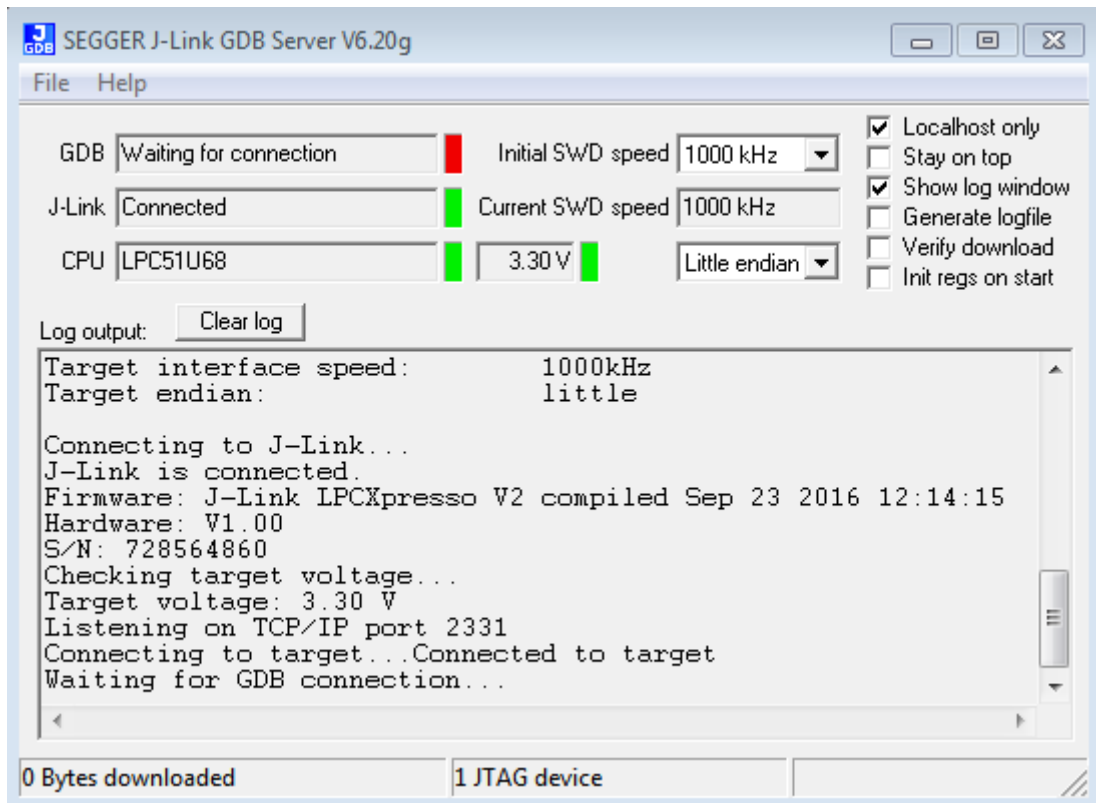
- Make sure that:
 - You have a standalone J-Link pod that is connected to the debug interface of your board. Note that some hardware platforms require hardware modification in order to function correctly with an external debug interface.

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

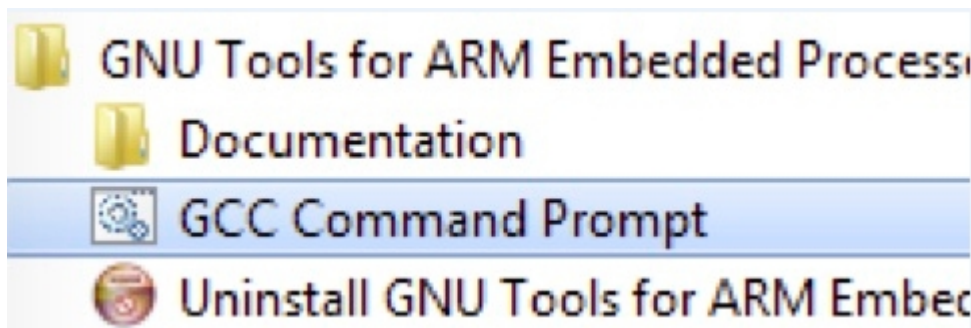
1. Connect the development platform to your PC via USB cable between the Link2 USB connector and the PC USB connector. If you are connecting for the first time, allow about 30 seconds for the devices to enumerate.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see Appendix A). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
 2. No parity
 3. 8 data bits
 4. 1 stop bit



3. 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".
4. Modify the settings as shown below. The target device selection chosen for this example is the LPC51U68
5. After it is connected, the screen should resemble this figure:



6. 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”.



7. 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:

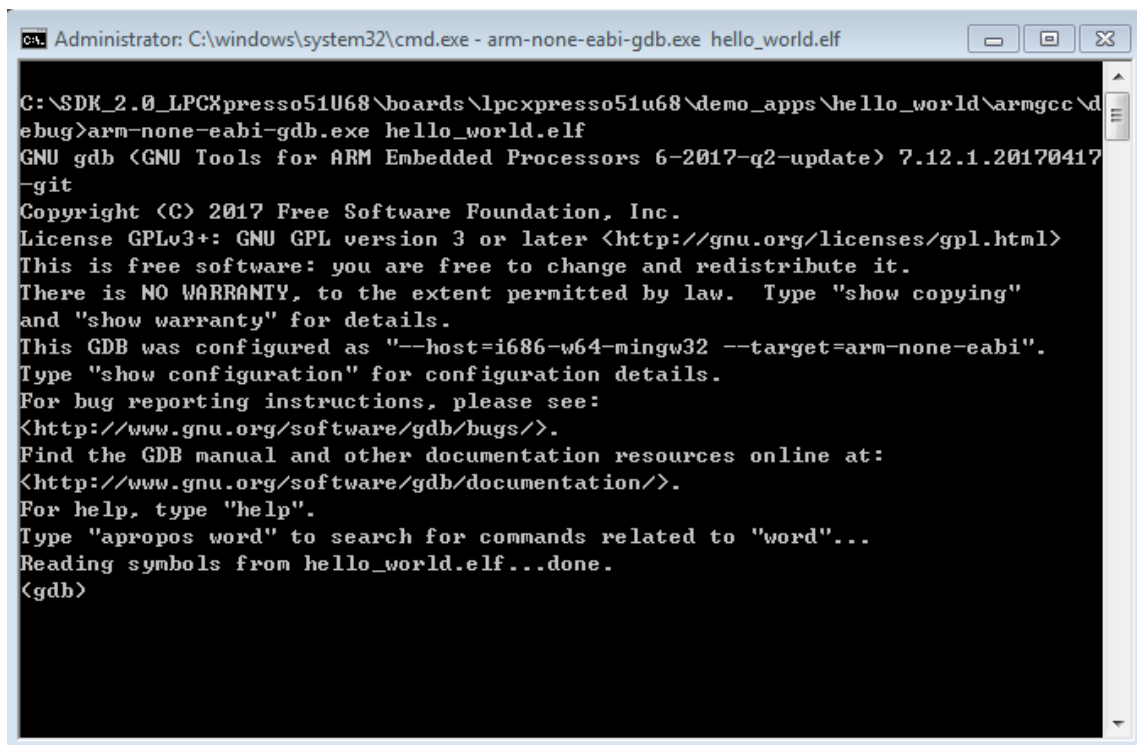
`<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc/debug`

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

For this example, the path is:

`<install_dir>/boards/lpcxpresso51U68/demo_apps/hello_world/armgcc/debug`

8. Run the command “arm-none-eabi-gdb.exe <application_name>.elf”. For this example, it is “arm-none-eabi-gdb.exe hello_world.elf”.



```

Administrator: C:\windows\system32\cmd.exe - arm-none-eabi-gdb.exe hello_world.elf

C:\SDK_2.0_LPCXpresso51U68\boards\lpcxpresso51u68\demo_apps\hello_world\armgcc\d
ebug>arm-none-eabi-gdb.exe hello_world.elf
GNU gdb (GNU Tools for ARM Embedded Processors 6-2017-q2-update) 7.12.1.20170417
-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello_world.elf...done.
(gdb)

```

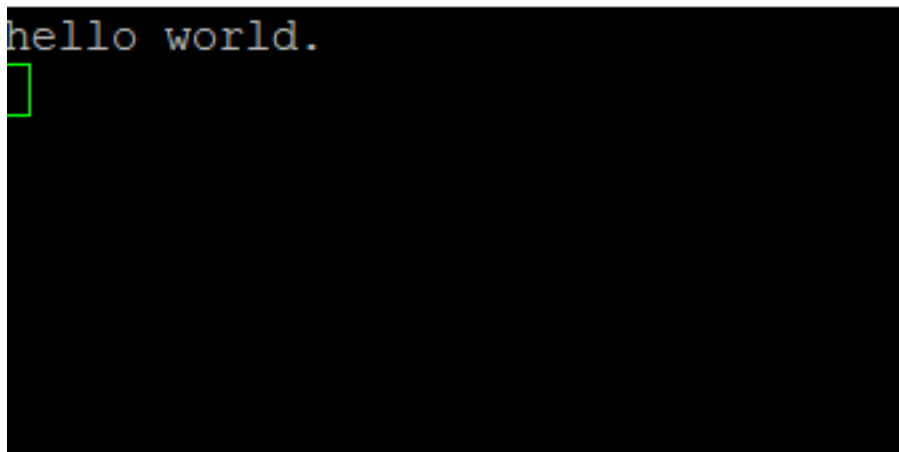
9. Run these commands:

1. "target remote localhost:2331"
2. "monitor reset"
3. "monitor go"
4. "monitor halt"
5. "load"
6. "monitor reg pc=(0x4)"
7. "monitor reg msp=(0x0)"

10. 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.

 COM3 - PuTTY



```

hello world.

```

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

Run a demo using MCUXpresso IDE

Note: Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

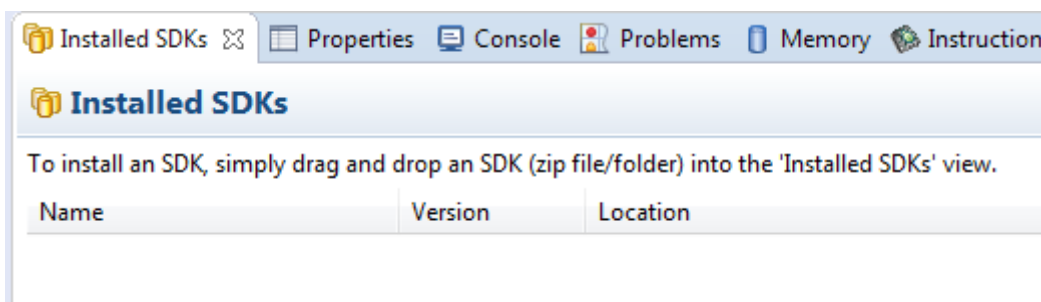
This section describes the steps required to configure MCUXpresso IDE to build, run, and debug example applications. The `hello_world` demo application targeted for the LPCXpresso51U68 hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

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

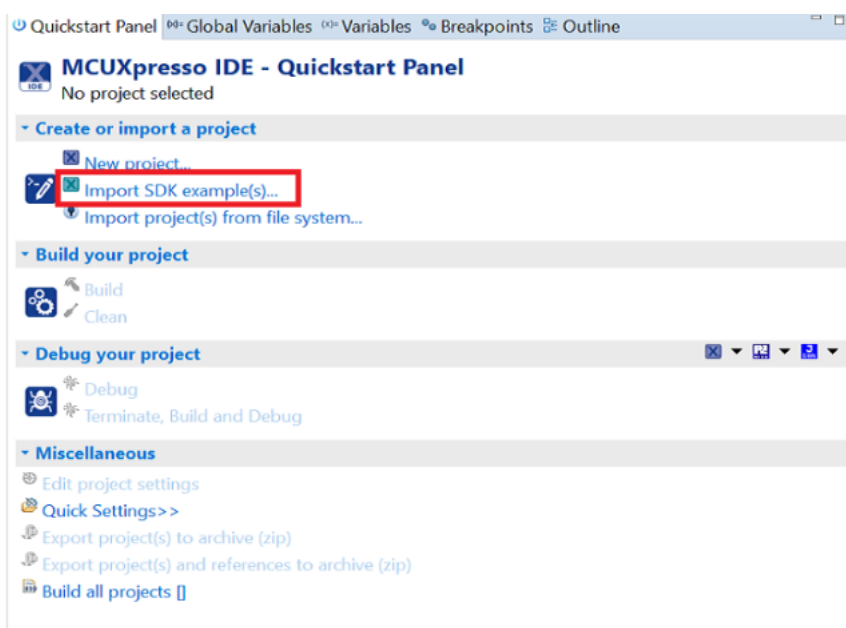
Parent topic: [Run a demo using MCUXpresso IDE](#)

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

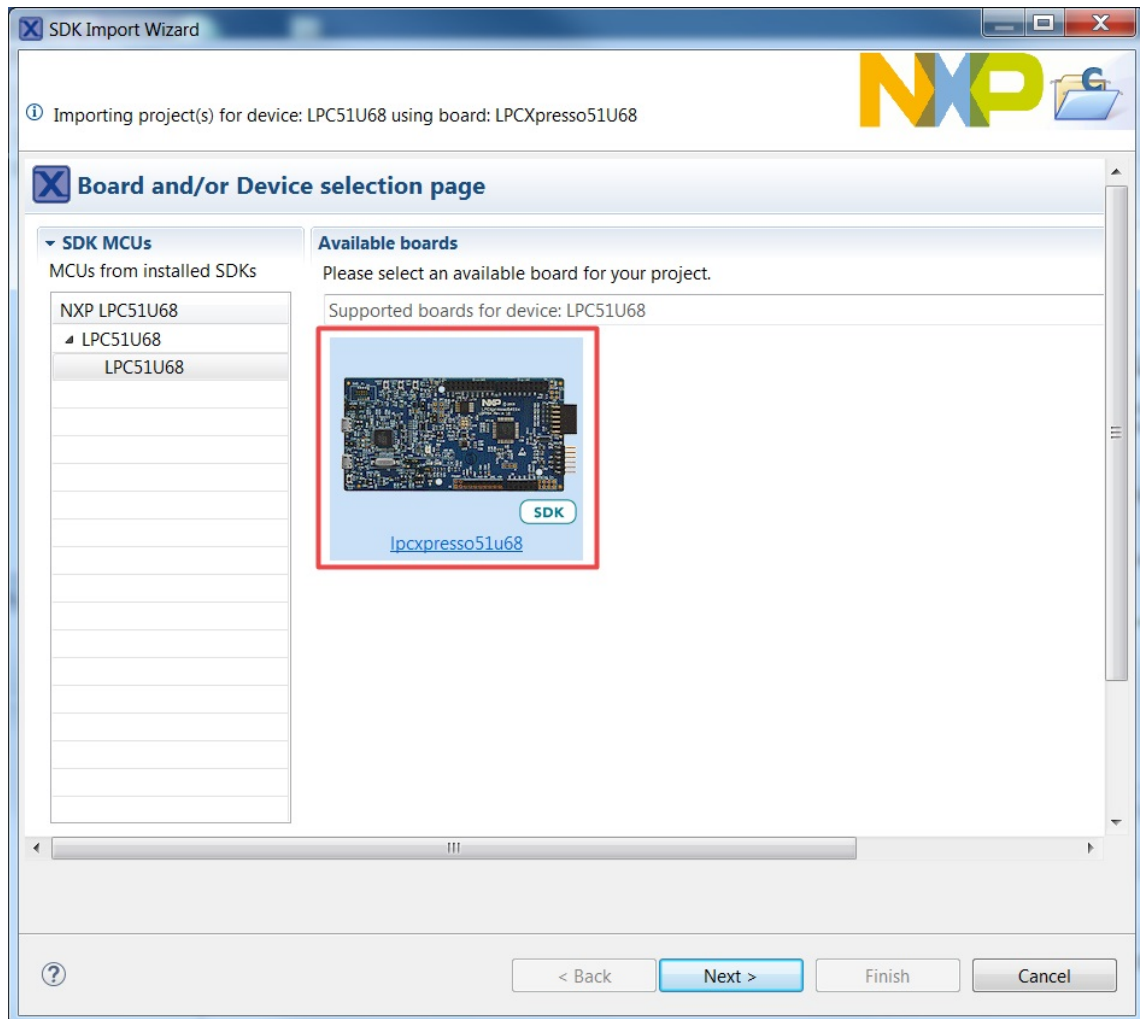
1. Drag and drop the SDK zip file into the “Installed SDKs” view to install an SDK. In the window that appears, click the “OK” button and wait until the import has finished.

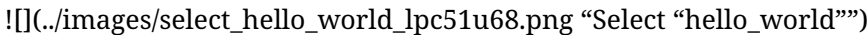
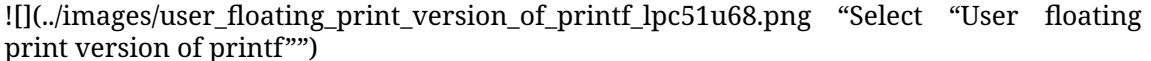


2. On the *Quickstart Panel*, click “Import SDK example(s)...”.



3. In the window that appears, expand the “LPC51U68” folder and select “LPC51U68”. Then, select “lpcxpresso51u68” and click the “Next” button.



4. Expand the “demo_apps” folder and select “hello_world”. Then, click the “Next” button.

5. Ensure the option “Redlib: Use floating point version of printf” is selected if the cases print floating point numbers on the terminal (for demo applications such as adc_basic, adc_burst, adc_dma, and adc_interrupt). Otherwise, there is no need to select it. Click the “Finish” button.


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

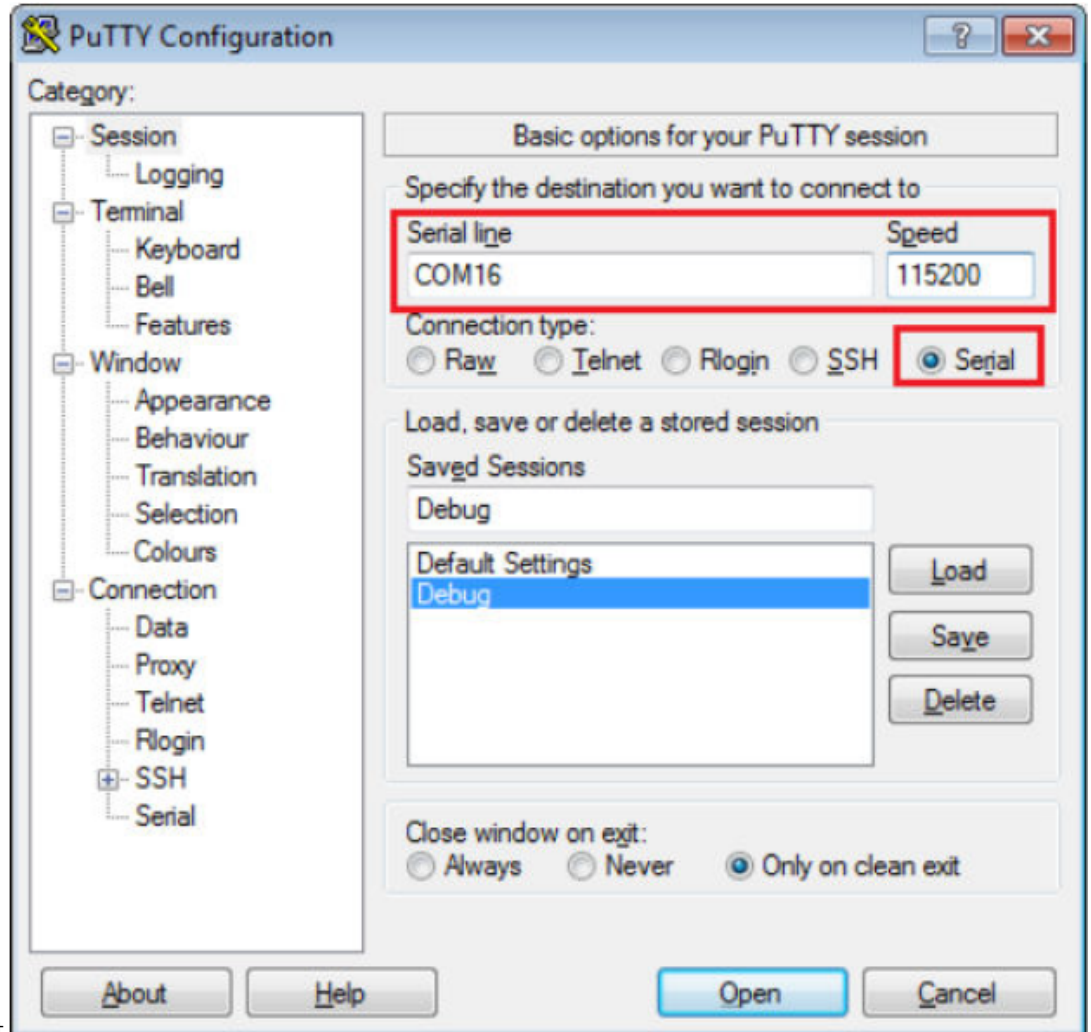
Run an example application For more information on debug probe support in the MCUXpresso IDE v11.0.0, visit community.nxp.com.

To download and run the application, perform these steps:

1. Reference the table in Appendix B to determine the debug interface that comes loaded on your specific hardware platform. For LPCXpresso boards, install the DFU jumper for the debug probe, then connect the debug probe USB connector.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see Appendix A). Configure

the terminal with these settings:

1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
2. No parity
3. 8 data bits

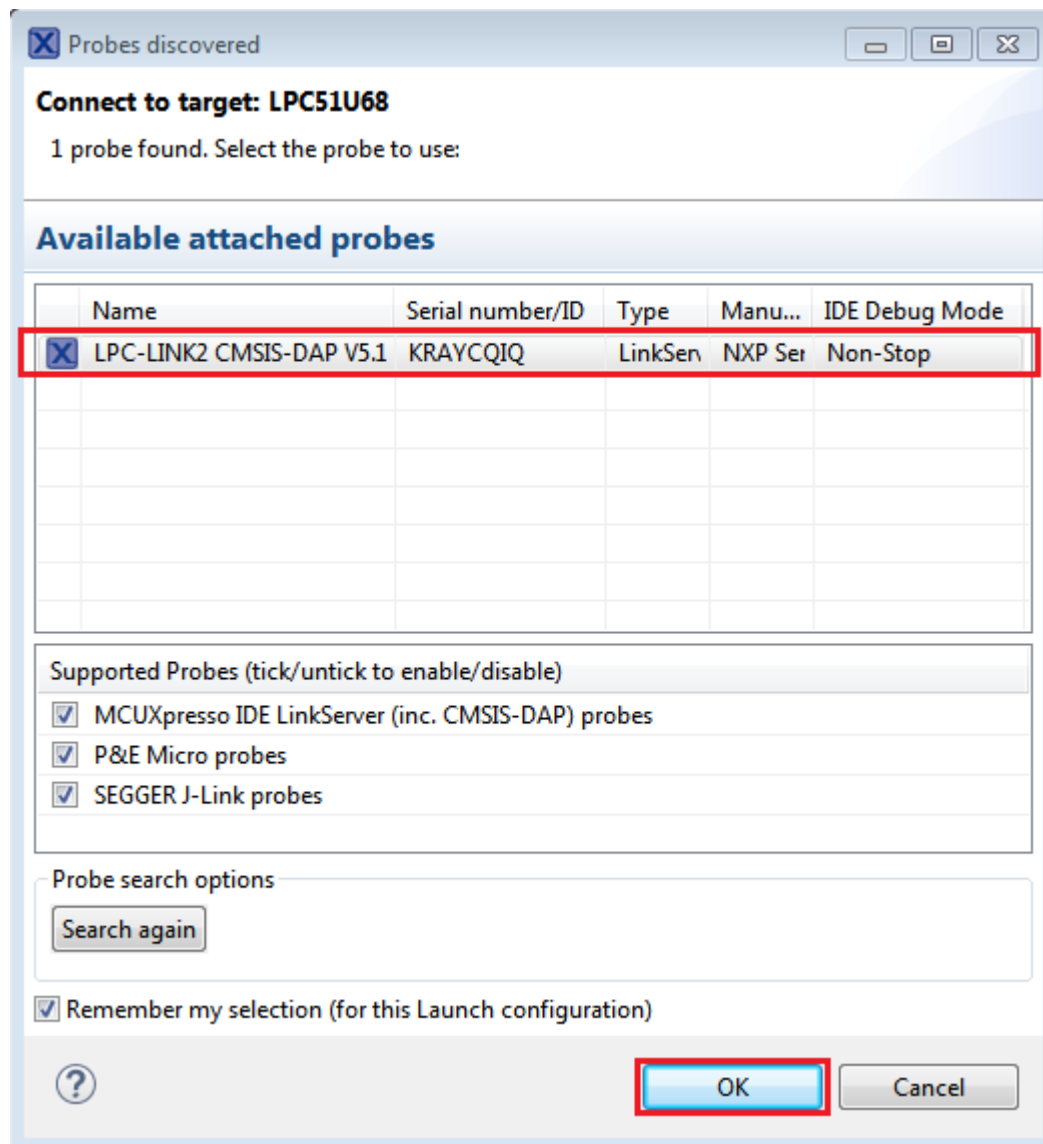


4. 1 stop bit

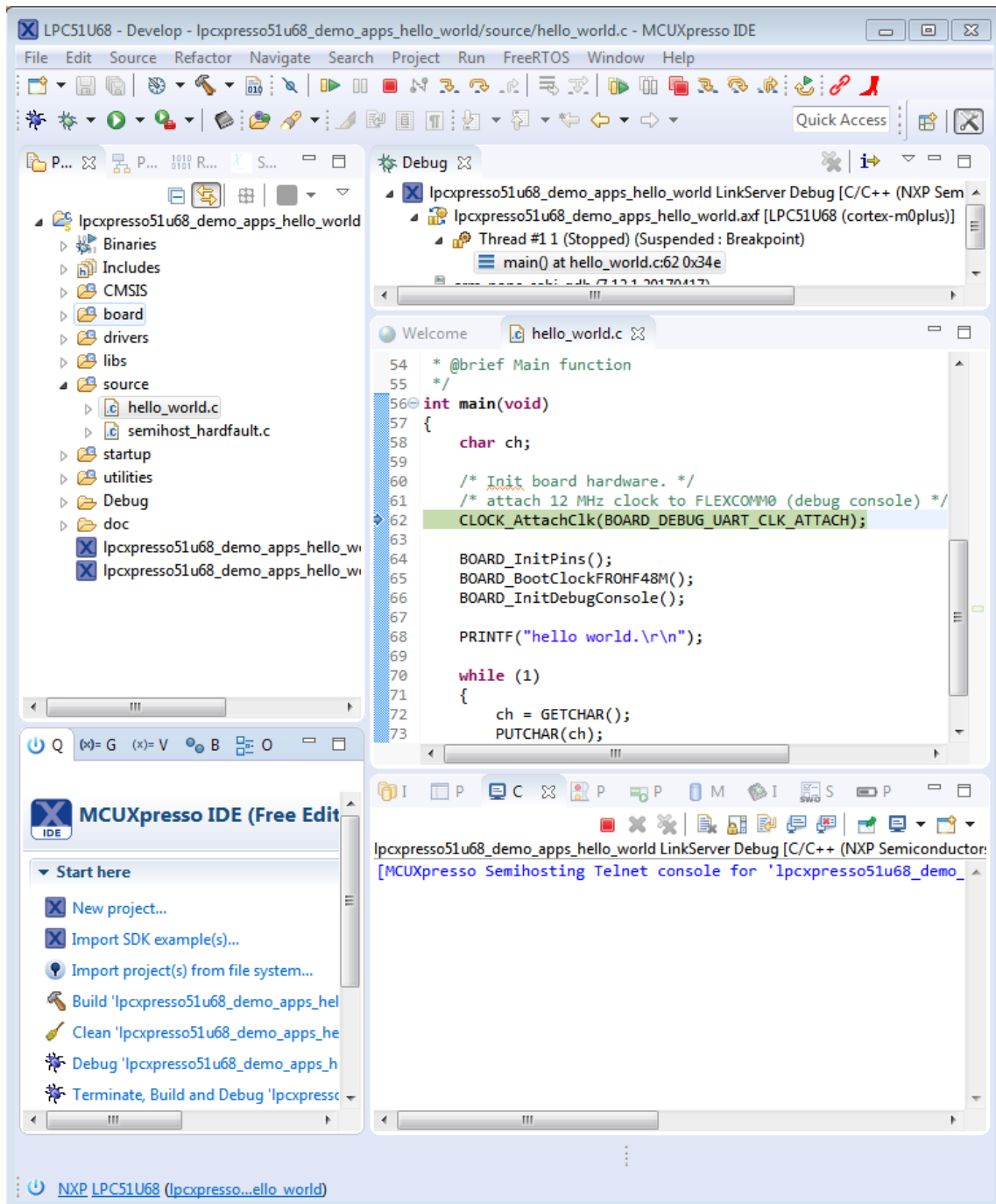
3. On the *Quickstart Panel*, click on “Debug ‘lpcxpresso51U68_demo_apps_hello_world’ [Debug]”.

![(./images/debug_hello_world_case_lpc51u68.png “Debug “hello_world” case”)]

4. The first time you debug a project, the Debug Emulator Selection Dialog is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click the “OK” button. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)



5. The application is downloaded to the target and automatically runs to main():



6. Start the application by clicking the “Resume” button.



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

 COM3 - PuTTY

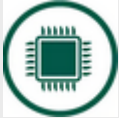
```
hello world.
```

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

MCUXpresso Config Tools

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

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

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

| **Clock tool** | For system clock configuration |



| **Peripherals tools** | For configuration of other peripherals |



| **TEE tool** | Configures access policies for memory area and peripherals helping to protect and

isolate sensitive parts of the application. |



| **Device Configuration tool** | Configures Device Configuration Data (DCD) contained in the program image that the Boot ROM code interprets to setup various on-chip peripherals prior the

program launch. |



MCUXpresso Config Tools can be accessed in the following products:

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

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

MCUXpresso IDE New Project Wizard

MCUXpresso IDE features a new project wizard. The wizard provides functionality for the user to create new projects from the installed SDKs (and from pre-installed part support). It offers user the flexibility to select and change multiple builds. The wizard also includes a library and provides source code options. The source code is organized as software components, categorized as drivers, utilities, and middleware.

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

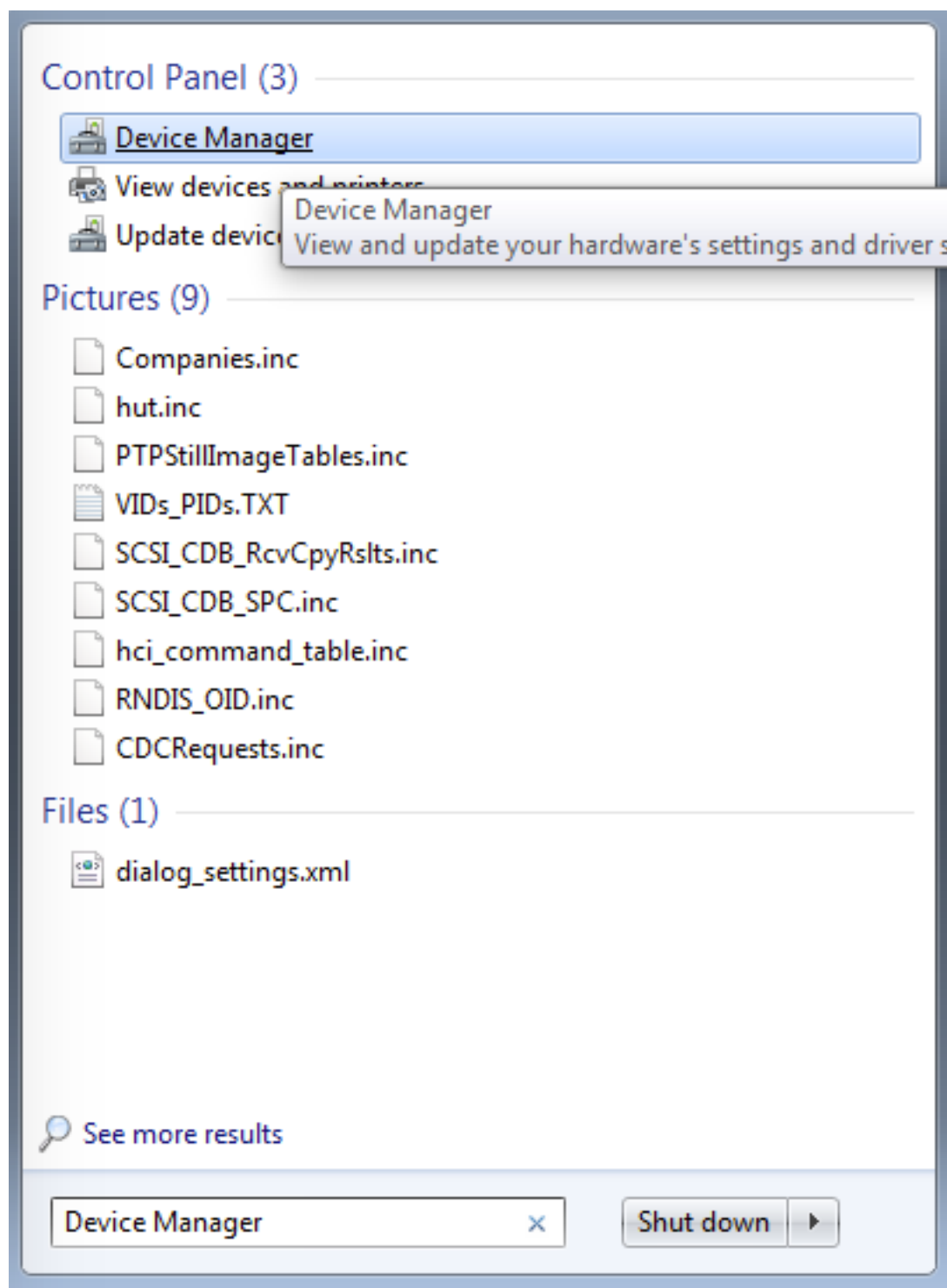


For more details and usage of new project wizard, see the *MCUXpresso_IDE_User_Guide.pdf* in the MCUXpresso IDE installation folder.

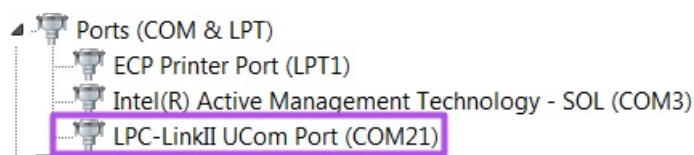
How to determine COM port

This section describes the steps necessary to determine the debug COM port number of your NXP hardware development platform. All NXP boards ship with a factory programmed, on-board debug interface, whether it's based on OpenSDA or the legacy P&E Micro OSJTAG interface. To determine what your specific board ships with, see [Default debug interfaces](#).

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. LPC-Link2



How to define IRQ handler in CPP files

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

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

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

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

Default debug interfaces

The MCUXpresso SDK supports various hardware platforms that come loaded with a variety of factory programmed debug interface configurations. *Table 1* lists the hardware platforms supported by the MCUXpresso SDK, their default debug interface, and any version information that helps differentiate a specific interface configuration.

Note: The *OpenSDA details* column in *Table 1* is not applicable to LPC.

Hardware platform	Default interface	OpenSDA details
EVK-MC56F83000	P&E Micro OSJTAG	N/A
EVK-MIMXRT595	CMSIS-DAP	N/A
EVK-MIMXRT685	CMSIS-DAP	N/A
FRDM-K22F	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.1
FRDM-K28F	DAPLink	OpenSDA v2.1
FRDM-K32L2A4S	CMSIS-DAP	OpenSDA v2.1
FRDM-K32L2B	CMSIS-DAP	OpenSDA v2.1
FRDM-K32W042	CMSIS-DAP	N/A
FRDM-K64F	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.0
FRDM-K66F	J-Link OpenSDA	OpenSDA v2.1
FRDM-K82F	CMSIS-DAP	OpenSDA v2.1
FRDM-KE15Z	DAPLink	OpenSDA v2.1
FRDM-KE16Z	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.2
FRDM-KL02Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL03Z	P&E Micro OpenSDA	OpenSDA v1.0

continues on next page

Table 1 – continued from previous page

Hardware platform	Default interface	OpenSDA details
FRDM-KL25Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL26Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL27Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL28Z	P&E Micro OpenSDA	OpenSDA v2.1
FRDM-KL43Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL46Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL81Z	CMSIS-DAP	OpenSDA v2.0
FRDM-KL82Z	CMSIS-DAP	OpenSDA v2.0
FRDM-KV10Z	CMSIS-DAP	OpenSDA v2.1
FRDM-KV11Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KV31F	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KW24	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.1
FRDM-KW36	DAPLink	OpenSDA v2.2
FRDM-KW41Z	CMSIS-DAP/DAPLink	OpenSDA v2.1 or greater
Hexiwear	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.0
HVP-KE18F	DAPLink	OpenSDA v2.2
HVP-KV46F150M	P&E Micro OpenSDA	OpenSDA v1
HVP-KV11Z75M	CMSIS-DAP	OpenSDA v2.1
HVP-KV58F	CMSIS-DAP	OpenSDA v2.1
HVP-KV31F120M	P&E Micro OpenSDA	OpenSDA v1
JN5189DK6	CMSIS-DAP	N/A
LPC54018 IoT Module	N/A	N/A
LPCXpresso54018	CMSIS-DAP	N/A
LPCXpresso54102	CMSIS-DAP	N/A
LPCXpresso54114	CMSIS-DAP	N/A
LPCXpresso51U68	CMSIS-DAP	N/A
LPCXpresso54608	CMSIS-DAP	N/A
LPCXpresso54618	CMSIS-DAP	N/A
LPCXpresso54628	CMSIS-DAP	N/A
LPCXpresso54S018M	CMSIS-DAP	N/A
LPCXpresso55s16	CMSIS-DAP	N/A
LPCXpresso55s28	CMSIS-DAP	N/A
LPCXpresso55s69	CMSIS-DAP	N/A
MAPS-KS22	J-Link OpenSDA	OpenSDA v2.0
MIMXRT1170-EVK	CMSIS-DAP	N/A
TWR-K21D50M	P&E Micro OSJTAG	N/AOpenSDA v2.0
TWR-K21F120M	P&E Micro OSJTAG	N/A
TWR-K22F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K24F120M	CMSIS-DAP/mbd	OpenSDA v2.1
TWR-K60D100M	P&E Micro OSJTAG	N/A
TWR-K64D120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K64F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K65D180M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K65D180M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV10Z32	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K80F150M	CMSIS-DAP	OpenSDA v2.1
TWR-K81F150M	CMSIS-DAP	OpenSDA v2.1
TWR-KE18F	DAPLink	OpenSDA v2.1
TWR-KL28Z72M	P&E Micro OpenSDA	OpenSDA v2.1
TWR-KL43Z48M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KL81Z72M	CMSIS-DAP	OpenSDA v2.0
TWR-KL82Z72M	CMSIS-DAP	OpenSDA v2.0
TWR-KM34Z75M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KM35Z75M	DAPLink	OpenSDA v2.2
TWR-KV10Z32	P&E Micro OpenSDA	OpenSDA v1.0

continues on next page

Table 1 – continued from previous page

Hardware platform	Default interface	OpenSDA details
TWR-KV11Z75M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV31F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV46F150M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV58F220M	CMSIS-DAP	OpenSDA v2.1
TWR-KW24D512	P&E Micro OpenSDA	OpenSDA v1.0
USB-KW24D512	N/A External probe	N/A
USB-KW41Z	CMSIS-DAP\DAPlink	OpenSDA v2.1 or greater

Updating debugger firmware

Updating LPCXpresso board firmware The LPCXpresso hardware platform comes with a CMSIS-DAP-compatible debug interface (known as LPC-Link2). This firmware in this debug interface may be updated using the host computer utility called LPCScript. This typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to re-program the debug probe firmware.

Note: If MCUXpresso IDE is used and the jumper making DFUlink is installed on the board (JP5 on some boards, but consult the board user manual or schematic for specific jumper number), LPC-Link2 debug probe boots to DFU mode, and MCUXpresso IDE automatically downloads the CMSIS-DAP firmware to the probe before flash memory programming (after clicking **Debug**). Using DFU mode ensures most up-to-date/compatible firmware is used with MCUXpresso IDE.

NXP provides the LPCScript utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto LPC-Link2 or LPCXpresso boards. The utility can be downloaded from www.nxp.com/lpcutilities.

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in LPCScript user guide (www.nxp.com/lpcutilities, select **LPCScript**, and then the documentation tab).

1. Install the LPCScript utility.
2. Unplug the board's USB cable.
3. Make the DFU link (install the jumper labelled DFUlink).
4. Connect the probe to the host via USB (use Link USB connector).
5. Open a command shell and call the appropriate script located in the LPCScript installation directory (<LPCScript install dir>).
 1. To program CMSIS-DAP debug firmware: <LPCScript install dir>/scripts/program_CMSIS
 2. To program J-Link debug firmware: <LPCScript install dir>/scripts/program_JLINK
6. Remove DFU link (remove the jumper installed in Step 3).
7. Re-power the board by removing the USB cable and plugging it in again.

Parent topic: [Updating debugger firmware](#)

1.3 Getting Started with MCUXpresso SDK GitHub

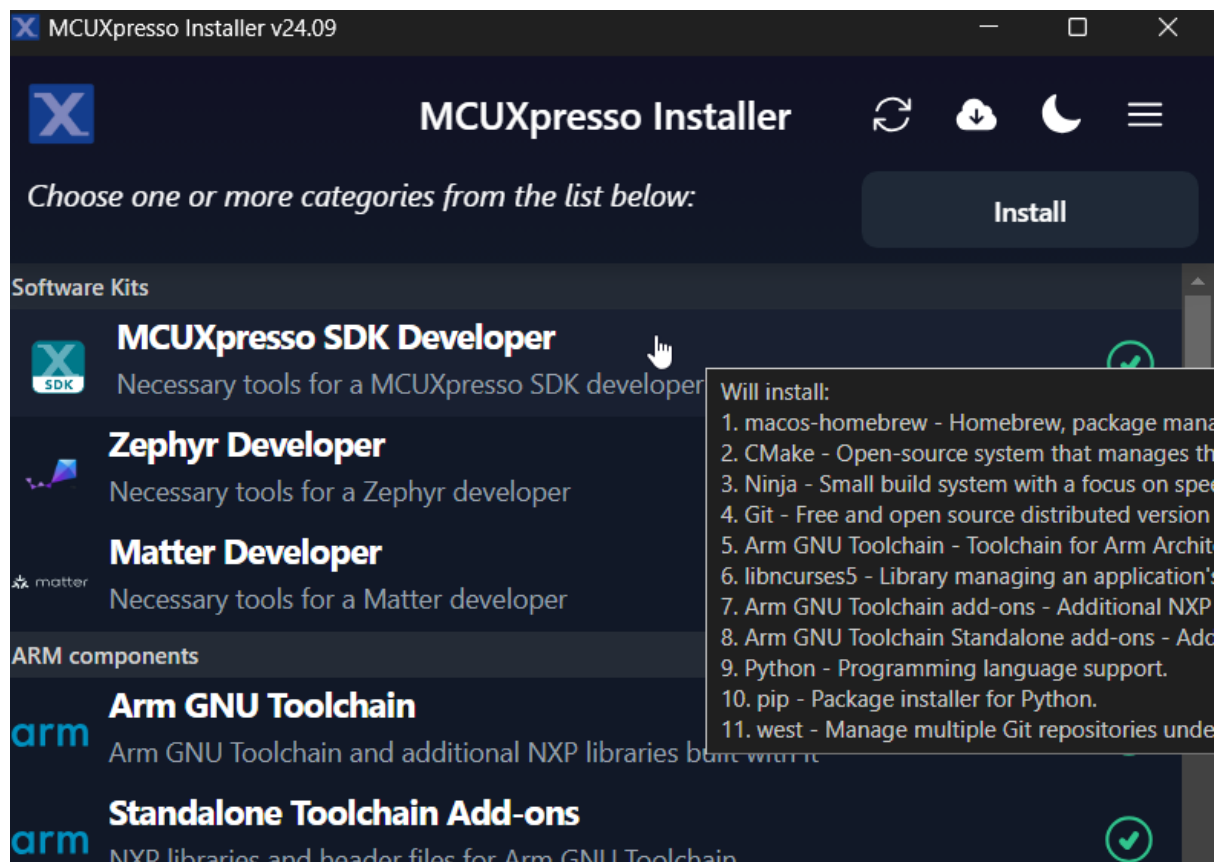
1.3.1 Getting Started with MCUXpresso SDK Repository

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.

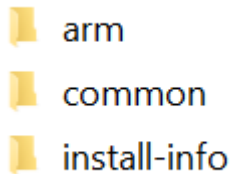
Toolchain MCUXpresso SDK supports all mainstream toolchains for embedded development. You can install your used or interested toolchains following the guides.

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

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



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

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

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

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

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

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

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

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

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

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

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

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

Get MCUXpresso SDK Repo

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

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

# Update the west projects
cd mcuxpresso-sdk
west update

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

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

1. Navigate to the core directory:

```
cd mcuxsdk
```

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

```
python -m venv .venv

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

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

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

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

3. Install the required Python packages:

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

Explore Contents

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

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

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

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

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

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

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

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

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

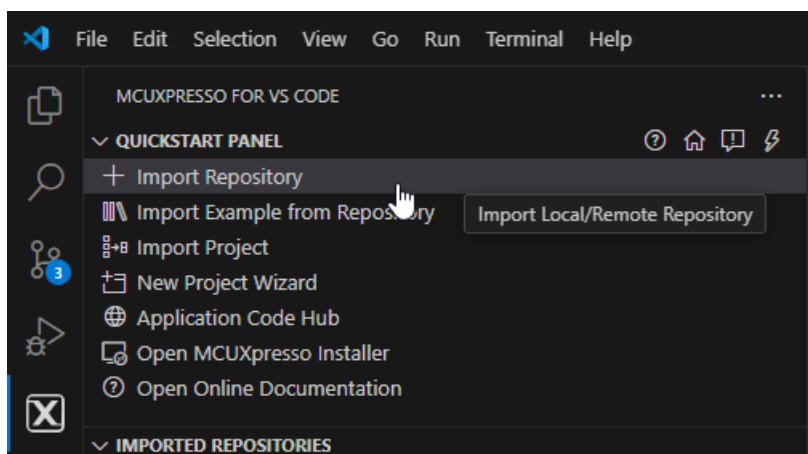
Run a demo using MCUXpresso for VS Code

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

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

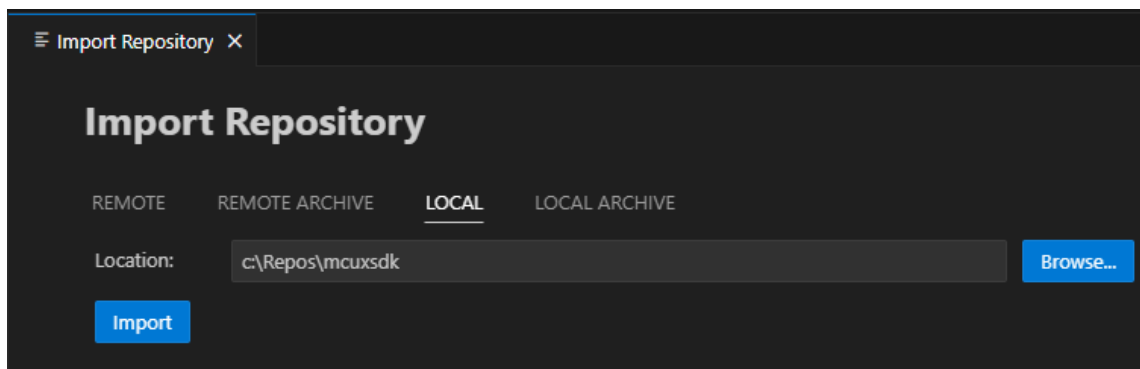
To build an example application:

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

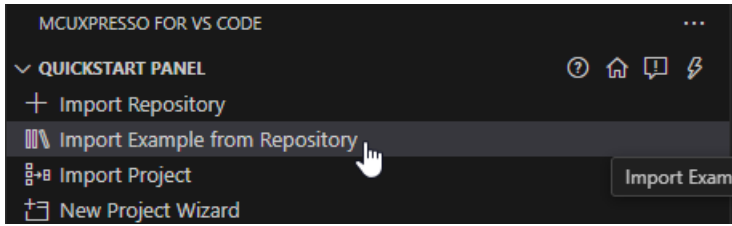


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

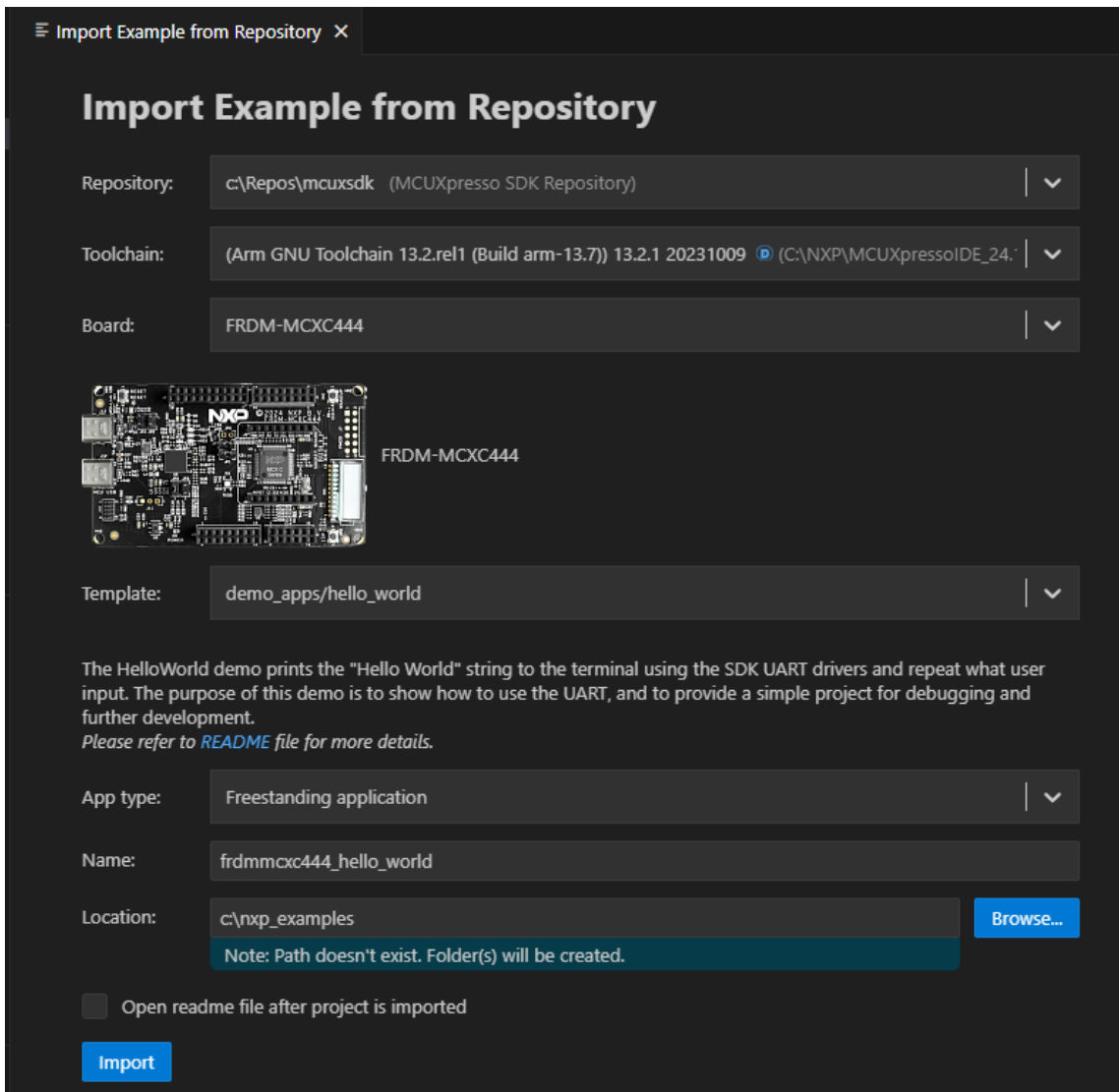
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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

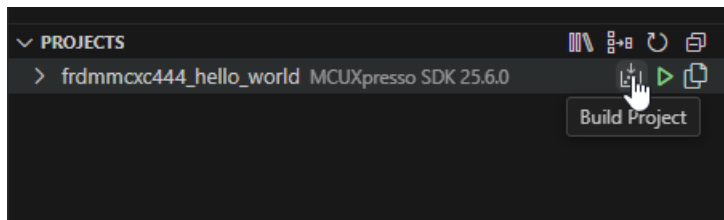


In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.

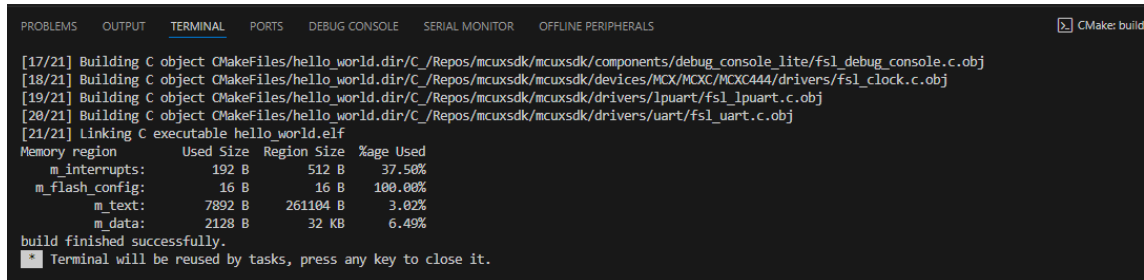


Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Free-standing applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Free-standing examples can be imported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

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

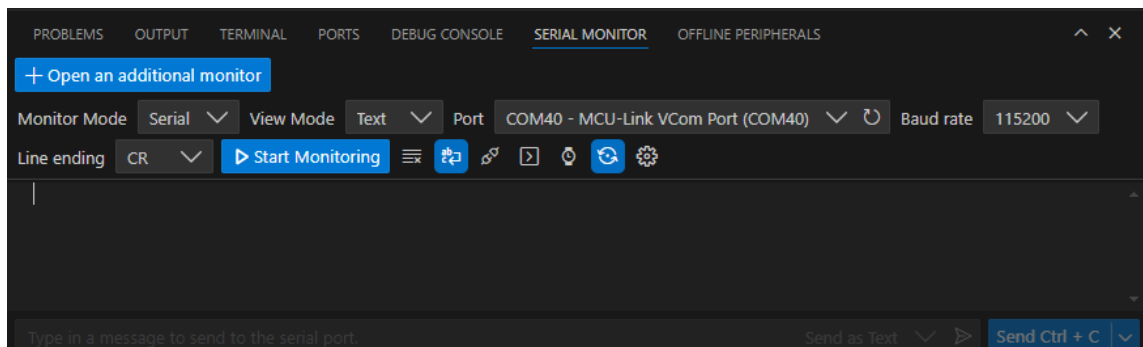


The integrated terminal will open at the bottom and will display the build output.

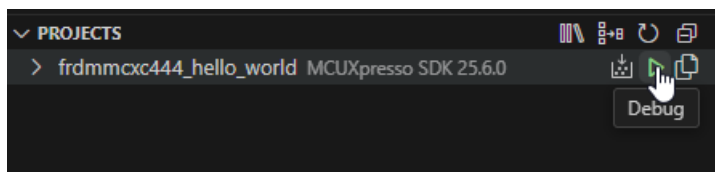


Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

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 [ ] [ ] [ ] [ ] [ ] [ ]
---- Opened the serial port COM40 ----
hello world.
|

```

Running a demo using ARMGCC CLI/IAR/MDK

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

```

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

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

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

Here are some typical usages for generating a SDK example:

```

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

For more details, please refer to System build.

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

1. Run cmake configuration

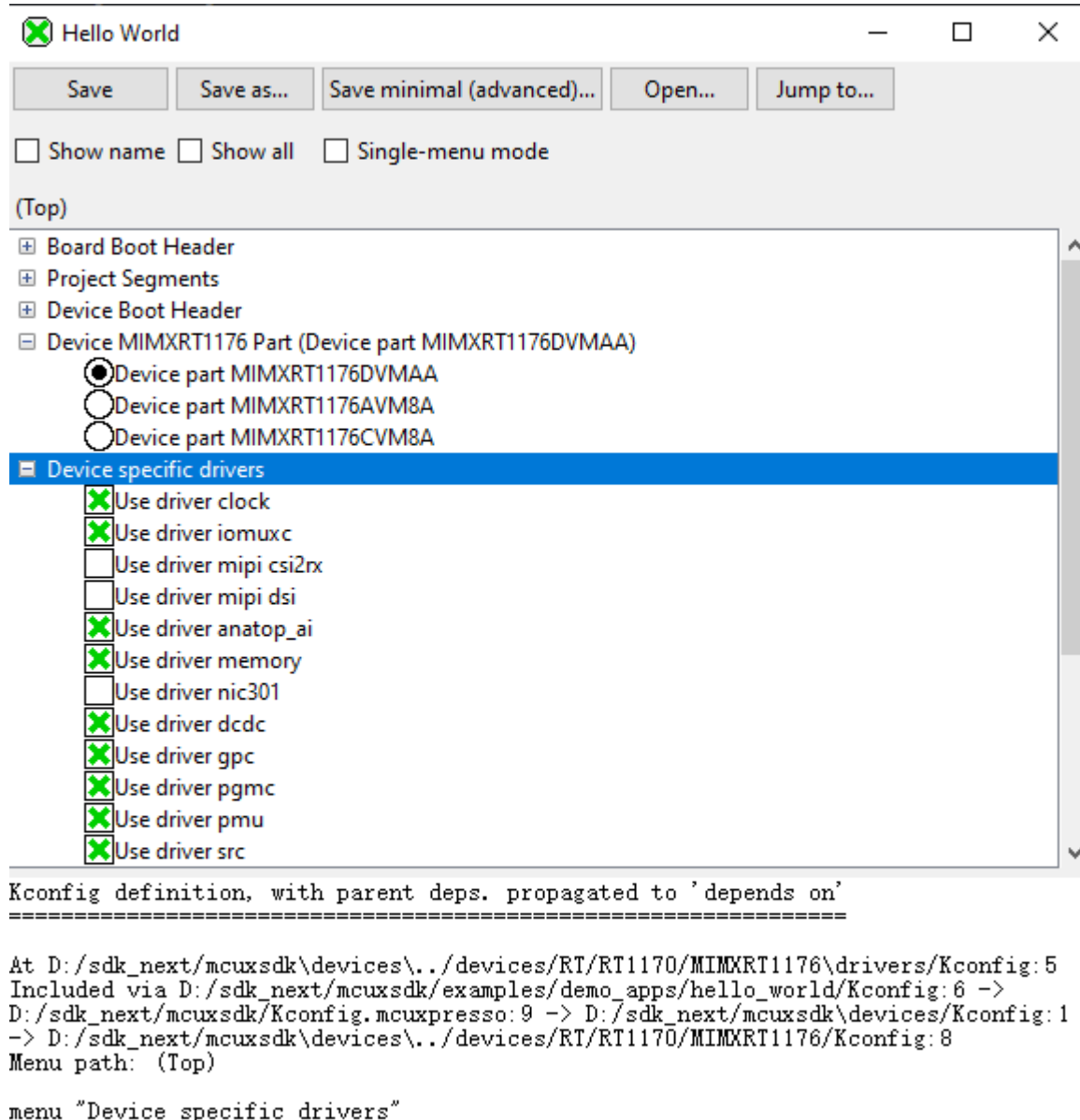
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

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

2. Run guiconfig target

```
west build -t guiconfig
```

Then you will get the Kconfig GUI launched, like



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

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

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

```
west flash -r linkserver
```

Debug Start a gdb interface by following command:

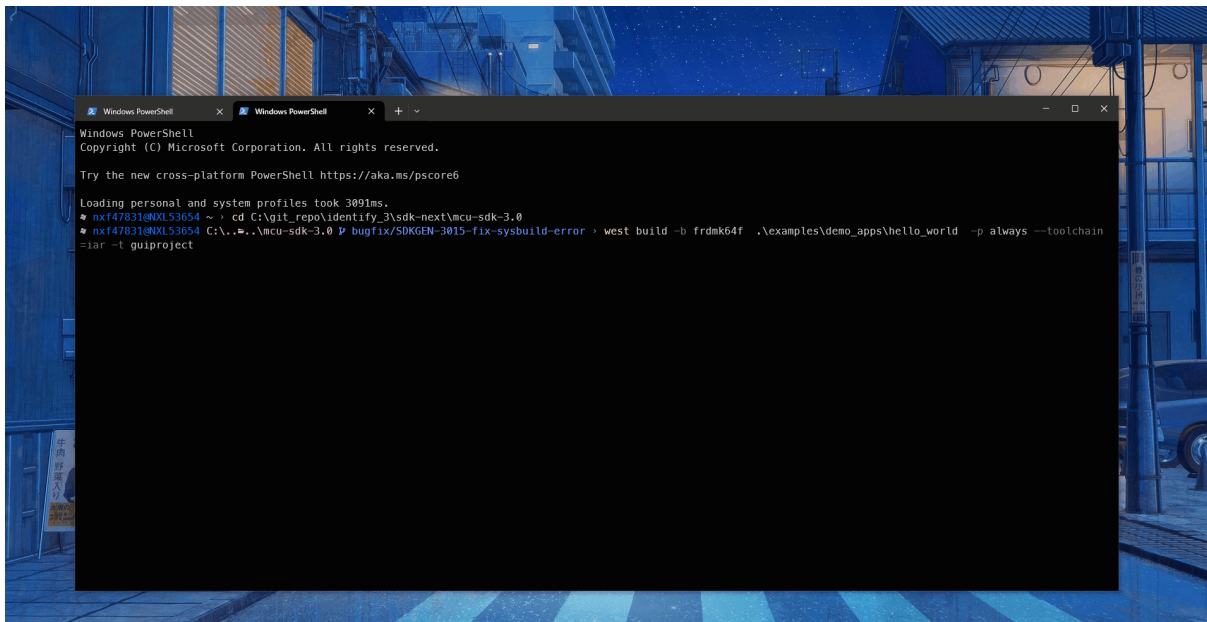
```
west debug -r linkserver
```

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

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

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

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



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

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

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

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

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

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

MCUXpresso SDK

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

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

Development tools

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

- MCUXpresso IDE, Rev. 25.06.xx
- IAR Embedded Workbench for Arm, version is 9.60.4
- Keil MDK, version is 5.41
- MCUXpresso for VS Code v25.06
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

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

Development boards	MCU devices
LPCXpresso51U68	LPC51U68JBD48, LPC51U68JBD64

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.

coreHTTP coreHTTP

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

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

TinyCBOR Concise Binary Object Representation (CBOR) Library

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

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

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

Release contents

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

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

Known issues

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

Cannot add SDK components into FreeRTOS projects

It is not possible to add any SDK components into FreeRTOS project using the MCUXpresso IDE New Project wizard.

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
-

LPC_ADC

[2.6.0]

- New Features
 - Added new feature macro to distinguish whether the GPADC_CTRL0_GPADC_TSAMP control bit is on the device.
 - Added new variable extendSampleTimeNumber to indicate the ADC extend sample time.
- Bugfix
 - Fixed the bug that incorrectly sets the PASS_ENABLE bit based on the sample time setting.

[2.5.3]

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

[2.5.2]

- Improvements
 - Integrated different sequence's sample time numbers into one variable.
- Bug Fixes
 - Fixed violation of MISRA C-2012 rule 20.9 .

[2.5.1]

- Bug Fixes
 - Fixed ADC conversion sequence priority misconfiguration issue in the ADC_SetConvSeqAHighPriority() and ADC_SetConvSeqBHighPriority() APIs.
- Improvements
 - Supported configuration ADC conversion sequence sampling time.

[2.5.0]

- Improvements
 - Add missing parameter tag of ADC_DoOffsetCalibration().
- Bug Fixes
 - Removed a duplicated API with typo in name: ADC_EnableShresholdCompareInterrupt().

[2.4.1]

- Bug Fixes
 - Enabled self-calibration after clock divider be changed to make sure the frequency update be taken.

[2.4.0]

- New Features
 - Added new API ADC_DoOffsetCalibration() which supports a specific operation frequency.
- Other Changes
 - Marked the ADC_DoSelfCalibration(ADC_Type *base) as deprecated.
- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.1 10.3 10.4 10.7 10.8 17.7.

[2.3.2]

- Improvements
 - Added delay after enabling using the ADC_GPADC_CTRL0_LDO_POWER_EN bit for JN5189/QN9090.
- New Features
 - Added support for platforms which have only one ADC sequence control/result register.

[2.3.1]

- Bug Fixes
 - Avoided writing ADC STARTUP register in ADC_Init().
 - Fixed Coverity zero divider error in ADC_DoSelfCalibration().

[2.3.0]

- Improvements
 - Updated “ADC_Init()” “ADC_GetChannelConversionResult()” API and “adc_resolution_t” structure to match QN9090.
 - Added “ADC_EnableTemperatureSensor” API.

[2.2.1]

- Improvements
 - Added a brief delay in uSec after ADC calibration start.

[2.2.0]

- Improvements
 - Updated “ADC_DoSelfCalibration” API and “adc_config_t” structure to match LPC845.

[2.1.0]

- Improvements
 - Renamed “ADC_EnableShresholdCompareInterrupt” to “ADC_EnableThresholdCompareInterrupt”.

[2.0.0]

- Initial version.
-
-

COMMON

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

CRC

[2.1.1]

- Fix MISRA issue.

[2.1.0]

- Add CRC_WriteSeed function.

[2.0.2]

- Fix MISRA issue.

[2.0.1]

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

[2.0.0]

- Initial version.
-

CTIMER

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.0]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPC_DMA

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.0]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Bug Fixes
 - Fixed Coverity issue.

[2.2.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

FLASHIAP

[2.0.6]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 5.6.

[2.0.5]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 9.1, 10.1 and 10.4.

[2.0.4]

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

[2.0.3]

- The FLASHIAP driver is marked as deprecated and will be removed in next release. All of its APIs are moved to the IAP driver. The names of FLASHIAP's APIs are updated from FLASHIAP_XXX() to IAP_XXX().

[2.0.2]

- Added the API for extended flash signature

[2.0.1]

- Removed two incorrect commands.

[2.0.0]

- Initial version.
-

FLEXCOMM**[2.0.2]**

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

FMEAS

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issues fixed: rule 10.4, rule 10.8.

[2.1.0]

- Updated “FMEAS_GetFrequency”, “FMEAS_StartMeasure”, “FMEAS_IsMeasureComplete” API and add definition to match ASYNC_SYSCON.

[2.0.0]

- Initial version ported from LPCOpen.
-

GINT

[2.1.1]

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

[2.1.0]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

GPIO

[2.1.7]

- Improvements
 - Enhanced GPIO_PinInit to enable clock internally.

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

- Improvements
 - Removed deprecated APIs.

[2.1.1]

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

[2.1.0]

- New Features
 - Added GPIO initialize API.

[2.0.0]

- Initial version.
-

I2C

[2.3.3]

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

[2.3.2]

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

[2.3.1]

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

[2.3.0]

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

[2.2.0]

- New Features
 - Added enumeration `_i2c_status_flags` to include all previous master and slave status flags, and added missing status flags.
 - Modified I2C_GetStatusFlags to get all I2C flags.
 - Added API I2C_ClearStatusFlags to clear all clearable flags not just master flags.
 - Modified master transactional APIs to enable bus event timeout interrupt during transfer, to avoid glitch on bus causing transfer hangs indefinitely.

- Bug Fixes
 - Fixed bug that status flags and interrupt enable masks share the same enumerations by adding enumeration `_i2c_interrupt_enable` for all master and slave interrupt sources.

[2.1.0]

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

[2.0.8]

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

[2.0.7]

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

[2.0.6]

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

[2.0.5]

- Bug Fixes
 - Fixed wrong assignment for datasize in I2C_InitTransferStateMachineDMA.
 - Fixed wrong working flow in I2C_RunTransferStateMachineDMA to ensure master can work in no start flag and no stop flag mode.
 - Fixed wrong working flow in I2C_RunTransferStateMachine and added kReceive-DataBeginState in _i2c_transfer_states to ensure master can work in no start flag and no stop flag mode.
 - Fixed wrong handle state in I2C_MasterTransferDMAHandleIRQ. After all the data has been transfered or nak is returned, handle state should be changed to idle.
- Improvements
 - Rounded up the calculated divider value in I2C_MasterSetBaudRate.

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.

I2S

[2.3.2]

- Bug Fixes
 - Fixed warning for comparison between pointer and integer.

[2.3.1]

- Bug Fixes
 - Updated the value of TX/RX software transfer state machine after transfer contents are submitted to avoid race condition.

[2.3.0]

- Improvements
 - Added api I2S_InstallDMADescriptorMemory/I2S_TransferSendLoopDMA/I2S_TransferReceiveLoopDMA to support loop transfer.
 - Added api I2S_EmptyTxFifo to support blocking flush tx fifo.
 - Updated api I2S_TransferAbortDMA by removed the blocking flush tx fifo from this function.
- Bug Fixes
 - Removed the while loop in abort transfer function to fix the dead loop issue under specific user case.

[2.2.2]

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

[2.2.1]

- Improvements
 - Added feature FSL_FEATURE_FLEXCOMM_INSTANCE_I2S_SUPPORT_SECONDARY_CHANNELn for the SOC has parts of instance support secondary channel.
- Bug Fixes
 - Added volatile statement for the state variable of i2s_handle and enable the mainline channel pair before enable interrupt to avoid the issue of code execution reordering which may cause the interrupt generated unexpectedly.

[2.2.0]

- Improvements
 - Added 8/16/24 bits mono data format transfer support in I2S driver.
 - Added new apis I2S_SetBitClockRate.
- Bug Fixes
 - Fixed the PA082 build warning.
 - Fixed the sign-compare warning.

- Fixed violations of the MISRA C-2012 rules 10.4, 10.8, 11.9, 10.1, 11.3, 13.5, 11.8, 10.3, 10.7.
- Fixed the Operand don't affect result Coverity issue.

[2.1.0]

- Improvements
 - Added a feature for the FLEXCOMM which supports I2S and has interconnection with DMIC.
 - Used a feature to control PDMDATA instead of I2S_CFG1_PDMDATA.
 - Added member bytesPerFrame in i2s_dma_handle_t, used for DMA transfer width configure, instead of using sizeof(uint32_t) hardcoded.
 - Used the macro provided by DMA driver to define the I2S DMA descriptor.
- Bug Fixes
 - Fixed the issue that I2S DMA driver always generated duplicate callback.

[2.0.3]

- New Features
 - Added a feature to remove configuration for the second channel on LPC51U68.

[2.0.2]

- New Features
 - Added ENABLE_IRQ handle after register I2S interrupt handle.

[2.0.1]

- Improvements
 - Unified the component full name to FLEXCOMM I2S (DMA) driver.

[2.0.0]

- Initial version.
-

I2S_DMA

[2.3.3]

- Bug Fixes
 - Fixed data size limit does not match the macro DMA_MAX_TRANSFER_BYTES issue.

[2.3.2]

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

[2.3.1]

- Refer I2S driver change log 2.0.1 to 2.3.1
-

IAP**[2.0.7]**

- Bug Fixes
 - Fixed IAP_ReinvokeISP bug that can't support UART ISP auto baud detection.

[2.0.6]

- Bug Fixes
 - Fixed IAP_ReinvokeISP wrong parameter setting.

[2.0.5]

- New Feature
 - Added support config flash memory access time.

[2.0.4]

- Bug Fixes
 - Fixed the violations of MISRA 2012 rules 9.1

[2.0.3]

- New Features
 - Added support for LPC 845's FAIM operation.
 - Added support for LPC 80x's fixed reference clock for flash controller.
 - Added support for LPC 5411x's Read UID command useless situation.
- Improvements
 - Improved the document and code structure.
- Bug Fixes
 - Fixed the violations of MISRA 2012 rules:
 - * Rule 10.1 10.3 10.4 17.7

[2.0.2]

- New Features
 - Added an API to read generated signature.
- Bug Fixes
 - Fixed the incorrect board support of IAP_ExtendedFlashSignatureRead().

[2.0.1]

- New Features
 - Added an API to read factory settings for some calibration registers.
- Improvements
 - Updated the size of result array in part APIs.

[2.0.0]

- Initial version.
-

INPUTMUX

[2.0.9]

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

[2.0.8]

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

[2.0.7]

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

[2.0.6]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Support channel mux setting in INPUTMUX_EnableSignal().

[2.0.0]

- Initial version.
-

IOCON**[2.2.0]**

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

MRT

[2.0.5]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

PINT

[2.2.0]

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

[2.1.13]

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

[2.1.12]

- Bug Fixes
 - Fixed coverity issue.

[2.1.11]

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

[2.1.10]

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

[2.1.9]

- Bug Fixes
 - Fixed MISRA-2012 rule 8.4.

[2.1.8]

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

[2.1.7]

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

[2.1.6]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-
-
-

RTC**[2.2.0]**

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

- * RTC_StartTimer
- * RTC_StopTimer
- * RTC_EnableInterrupts
- * RTC_DisableInterrupts
- * RTC_GetEnabledInterrupts

[2.0.0]

- Initial version.
-

SCTIMER

[2.5.1]

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

[2.5.0]

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

[2.4.9]

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

[2.4.8]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

- Bug Fixes
 - Fix SCT_EV_STATE_STATEMSKn macro build error.

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]**[2.3.0]**

- Bug Fixes
 - Fixed the potential overflow issue of pulseperiod variable in SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle API.
 - Fixed the issue of SCTIMER_CreateAndScheduleEvent API does not correctly work with 32 bit unified counter.
 - Fixed the issue of position of clear counter operation in SCTIMER_Init API.
- Improvements
 - Update SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle to support generate 0% and 100% PWM signal.
 - Add SCTIMER_SetupEventActiveDirection API to configure event activity direction.
 - Update SCTIMER_StartTimer/SCTIMER_StopTimer API to support start/stop low counter and high counter at the same time.
 - Add SCTIMER_SetCounterState/SCTIMER_GetCounterState API to write/read counter current state value.
 - Update APIs to make it meaningful.
 - * SCTIMER_SetEventInState
 - * SCTIMER_ClearEventInState

- * SCTIMER_GetEventInState

[2.2.0]

- Improvements
 - Updated for 16-bit register access.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

SPI**[2.3.2]**

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

[2.3.1]

- Improvements
 - Changed SPI_DUMMYDATA to 0x00.

[2.3.0]

- Update version.

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

SPI_DMA

[2.2.1]

- Bug Fixes
 - Fixed MISRA 2012 11.6 issue..

[2.2.0]

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

USART

[2.8.5]

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

[2.8.4]

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

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

- New Features
 - Added the rx timeout interrupts and status flags of bus status.
 - Added new rx timeout configuration item in usart_config_t.
 - Added API USART_SetRxTimeoutConfig for rx timeout configuration.
- Improvements

- When the calculated baudrate cannot meet user's configuration, lower OSR value is allowed to use.

[2.7.0]

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

[2.6.0]

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

[2.5.0]

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

[2.4.0]

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

[2.3.1]

- Bug Fixes
 - Fixed bug that operation on `INTENSET`, `INTENCLR`, `FIFOINTENSET` and `FIFOINTENCLR` should use bitwise operation not 'or' operation.
 - Fixed bug that if rx interrupt occurs before TX interrupt is enabled and after `txDataSize` is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.

- Improvements
 - Added check for baud rate's accuracy that returns `kStatus_USART_BaudrateNotSupport` when the best achieved baud rate is not within 3% error of configured baud rate.

[2.3.0]

- New Features
 - Added APIs to configure 9-bit data mode, set slave address and send address.
 - Modified `USART_TransferReceiveNonBlocking` and `USART_TransferHandleIRQ` to use 9-bit mode in multi-slave system.

[2.2.0]

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

[2.1.1]

- Improvements
 - Added check for transmitter idle in `USART_TransferHandleIRQ` and `USART_TransferSendDMACallback` to ensure all the data would be sent out to bus.
 - Modified `USART_ReadBlocking` so that if more than one receiver errors occur, all status flags will be cleared and the most severe error status will be returned.
- Bug Fixes
 - Eliminated IAR Pa082 warnings.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.

[2.1.0]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

USART_DMA

[2.6.0]

- Refer USART driver change log 2.0.1 to 2.6.0
-

UTICK

[2.0.5]

- Improvements
 - Improved for SOC RW610.

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

WWDT**[2.1.9]**

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

[2.1.8]

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

[2.1.7]

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

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

1.6 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.

[LPC51U68](#)

1.7 Middleware Documentation

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

1.7.1 FreeMASTER

freemaster

1.7.2 FreeRTOS

FreeRTOS

Chapter 2

LPC51U68

2.1 Clock Driver

enum _clock_ip_name

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

Values:

enumerator kCLOCK_IpInvalid

Invalid Ip Name.

enumerator kCLOCK_Rom

Clock gate name: Rom.

enumerator kCLOCK_Flash

Clock gate name: Flash.

enumerator kCLOCK_Fmc

Clock gate name: Fmc.

enumerator kCLOCK_InputMux

Clock gate name: InputMux.

enumerator kCLOCK_Iocon

Clock gate name: Iocon.

enumerator kCLOCK_Gpio0

Clock gate name: Gpio0.

enumerator kCLOCK_Gpio1

Clock gate name: Gpio1.

enumerator kCLOCK_Pint

Clock gate name: Pint.

enumerator kCLOCK_Gint

Clock gate name: Gint, GPIO_GLOBALINT0 and GPIO_GLOBALINT1 share the same slot

enumerator kCLOCK_Dma

Clock gate name: Dma.

enumerator kCLOCK_Crc

Clock gate name: Crc.

enumerator kCLOCK_Wwdt
Clock gate name: Wwdt.

enumerator kCLOCK_Rtc
Clock gate name: Rtc.

enumerator kCLOCK_Adc0
Clock gate name: Adc0.

enumerator kCLOCK_Mrt
Clock gate name: Mrt.

enumerator kCLOCK_Sct0
Clock gate name: Sct0.

enumerator kCLOCK_Utick
Clock gate name: Utick.

enumerator kCLOCK_FlexComm0
Clock gate name: FlexComm0.

enumerator kCLOCK_FlexComm1
Clock gate name: FlexComm1.

enumerator kCLOCK_FlexComm2
Clock gate name: FlexComm2.

enumerator kCLOCK_FlexComm3
Clock gate name: FlexComm3.

enumerator kCLOCK_FlexComm4
Clock gate name: FlexComm4.

enumerator kCLOCK_FlexComm5
Clock gate name: FlexComm5.

enumerator kCLOCK_FlexComm6
Clock gate name: FlexComm6.

enumerator kCLOCK_FlexComm7
Clock gate name: FlexComm7.

enumerator kCLOCK_MinUart0
Clock gate name: MinUart0.

enumerator kCLOCK_MinUart1
Clock gate name: MinUart1.

enumerator kCLOCK_MinUart2
Clock gate name: MinUart2.

enumerator kCLOCK_MinUart3
Clock gate name: MinUart3.

enumerator kCLOCK_MinUart4
Clock gate name: MinUart4.

enumerator kCLOCK_MinUart5
Clock gate name: MinUart5.

enumerator kCLOCK_MinUart6
Clock gate name: MinUart6.

enumerator kCLOCK_MinUart7
Clock gate name: MinUart7.

enumerator kCLOCK_LSpi0
Clock gate name: LSpi0.

enumerator kCLOCK_LSpi1
Clock gate name: LSpi1.

enumerator kCLOCK_LSpi2
Clock gate name: LSpi2.

enumerator kCLOCK_LSpi3
Clock gate name: LSpi3.

enumerator kCLOCK_LSpi4
Clock gate name: LSpi4.

enumerator kCLOCK_LSpi5
Clock gate name: LSpi5.

enumerator kCLOCK_LSpi6
Clock gate name: LSpi6.

enumerator kCLOCK_LSpi7
Clock gate name: LSpi7.

enumerator kCLOCK_BI2c0
Clock gate name: BI2c0.

enumerator kCLOCK_BI2c1
Clock gate name: BI2c1.

enumerator kCLOCK_BI2c2
Clock gate name: BI2c2.

enumerator kCLOCK_BI2c3
Clock gate name: BI2c3.

enumerator kCLOCK_BI2c4
Clock gate name: BI2c4.

enumerator kCLOCK_BI2c5
Clock gate name: BI2c5.

enumerator kCLOCK_BI2c6
Clock gate name: BI2c6.

enumerator kCLOCK_BI2c7
Clock gate name: BI2c7.

enumerator kCLOCK_FlexI2s0
Clock gate name: FlexI2s0.

enumerator kCLOCK_FlexI2s1
Clock gate name: FlexI2s1.

enumerator kCLOCK_FlexI2s2
Clock gate name: FlexI2s2.

enumerator kCLOCK_FlexI2s3
Clock gate name: FlexI2s3.

enumerator kCLOCK_FlexI2s4

Clock gate name: FlexI2s4.

enumerator kCLOCK_FlexI2s5

Clock gate name: FlexI2s5.

enumerator kCLOCK_FlexI2s6

Clock gate name: FlexI2s6.

enumerator kCLOCK_FlexI2s7

Clock gate name: FlexI2s7.

enumerator kCLOCK_Ct32b2

Clock gate name: Ct32b2.

enumerator kCLOCK_Usbd0

Clock gate name: Usbd0.

enumerator kCLOCK_Ctimer0

Clock gate name: Ctimer0.

enumerator kCLOCK_Ctimer1

Clock gate name: Ctimer1.

enumerator kCLOCK_Ctimer3

Clock gate name: Ctimer3.

enum _clock_name

Clock name used to get clock frequency.

Values:

enumerator kCLOCK_CoreSysClk

Core/system clock (aka MAIN_CLK)

enumerator kCLOCK_BusClk

Bus clock (AHB clock)

enumerator kCLOCK_ClockOut

CLOCKOUT

enumerator kCLOCK_FroHf

FRO48/96

enumerator kCLOCK_Fro12M

FRO12M

enumerator kCLOCK_ExtClk

External Clock

enumerator kCLOCK_PllOut

PLL Output

enumerator kCLOCK_WdtOsc

Watchdog Oscillator

enumerator kCLOCK_Frg

Frg Clock

enumerator kCLOCK_AsyncApbClk

Async APB clock

enumerator kCLOCK_FlexI2S
FlexI2S clock

enum _async_clock_src

Clock source selections for the asynchronous APB clock.

Values:

enumerator kCLOCK_AsyncMainClk
Main System clock

enumerator kCLOCK_AsyncFro12Mhz
12MHz FRO

enum _clock_attach_id

The enumerator of clock attach Id.

Values:

enumerator kFRO12M_to_MAIN_CLK
Attach FRO12M to MAIN_CLK.

enumerator kEXT_CLK_to_MAIN_CLK
Attach EXT_CLK to MAIN_CLK.

enumerator kWDT_OSC_to_MAIN_CLK
Attach WDT_OSC to MAIN_CLK.

enumerator kFRO_HF_to_MAIN_CLK
Attach FRO_HF to MAIN_CLK.

enumerator kSYS_PLL_to_MAIN_CLK
Attach SYS_PLL to MAIN_CLK.

enumerator kOSC32K_to_MAIN_CLK
Attach OSC32K to MAIN_CLK.

enumerator kFRO12M_to_SYS_PLL
Attach FRO12M to SYS_PLL.

enumerator kEXT_CLK_to_SYS_PLL
Attach EXT_CLK to SYS_PLL.

enumerator kWDT_OSC_to_SYS_PLL
Attach WDT_OSC to SYS_PLL.

enumerator kOSC32K_to_SYS_PLL
Attach OSC32K to SYS_PLL.

enumerator kNONE_to_SYS_PLL
Attach NONE to SYS_PLL.

enumerator kMAIN_CLK_to_ASYNC_APB
Attach MAIN_CLK to ASYNC_APB.

enumerator kFRO12M_to_ASYNC_APB
Attach FRO12M to ASYNC_APB.

enumerator kMAIN_CLK_to_ADC_CLK
Attach MAIN_CLK to ADC_CLK.

enumerator kSYS_PLL_to_ADC_CLK
Attach SYS_PLL to ADC_CLK.

enumerator kFRO_HF_to_ADC_CLK
Attach FRO_HF to ADC_CLK.

enumerator kNONE_to_ADC_CLK
Attach NONE to ADC_CLK.

enumerator kFRO12M_to_FLEXCOMM0
Attach FRO12M to FLEXCOMM0.

enumerator kFRO_HF_to_FLEXCOMM0
Attach FRO_HF to FLEXCOMM0.

enumerator kSYS_PLL_to_FLEXCOMM0
Attach SYS_PLL to FLEXCOMM0.

enumerator kMCLK_to_FLEXCOMM0
Attach MCLK to FLEXCOMM0.

enumerator kFRG_to_FLEXCOMM0
Attach FRG to FLEXCOMM0.

enumerator kNONE_to_FLEXCOMM0
Attach NONE to FLEXCOMM0.

enumerator kFRO12M_to_FLEXCOMM1
Attach FRO12M to FLEXCOMM1.

enumerator kFRO_HF_to_FLEXCOMM1
Attach FRO_HF to FLEXCOMM1.

enumerator kSYS_PLL_to_FLEXCOMM1
Attach SYS_PLL to FLEXCOMM1.

enumerator kMCLK_to_FLEXCOMM1
Attach MCLK to FLEXCOMM1.

enumerator kFRG_to_FLEXCOMM1
Attach FRG to FLEXCOMM1.

enumerator kNONE_to_FLEXCOMM1
Attach NONE to FLEXCOMM1.

enumerator kFRO12M_to_FLEXCOMM2
Attach FRO12M to FLEXCOMM2.

enumerator kFRO_HF_to_FLEXCOMM2
Attach FRO_HF to FLEXCOMM2.

enumerator kSYS_PLL_to_FLEXCOMM2
Attach SYS_PLL to FLEXCOMM2.

enumerator kMCLK_to_FLEXCOMM2
Attach MCLK to FLEXCOMM2.

enumerator kFRG_to_FLEXCOMM2
Attach FRG to FLEXCOMM2.

enumerator kNONE_to_FLEXCOMM2
Attach NONE to FLEXCOMM2.

enumerator kFRO12M_to_FLEXCOMM3
Attach FRO12M to FLEXCOMM3.

enumerator kFRO_HF_to_FLEXCOMM3
Attach FRO_HF to FLEXCOMM3.

enumerator kSYS_PLL_to_FLEXCOMM3
Attach SYS_PLL to FLEXCOMM3.

enumerator kMCLK_to_FLEXCOMM3
Attach MCLK to FLEXCOMM3.

enumerator kFRG_to_FLEXCOMM3
Attach FRG to FLEXCOMM3.

enumerator kNONE_to_FLEXCOMM3
Attach NONE to FLEXCOMM3.

enumerator kFRO12M_to_FLEXCOMM4
Attach FRO12M to FLEXCOMM4.

enumerator kFRO_HF_to_FLEXCOMM4
Attach FRO_HF to FLEXCOMM4.

enumerator kSYS_PLL_to_FLEXCOMM4
Attach SYS_PLL to FLEXCOMM4.

enumerator kMCLK_to_FLEXCOMM4
Attach MCLK to FLEXCOMM4.

enumerator kFRG_to_FLEXCOMM4
Attach FRG to FLEXCOMM4.

enumerator kNONE_to_FLEXCOMM4
Attach NONE to FLEXCOMM4.

enumerator kFRO12M_to_FLEXCOMM5
Attach FRO12M to FLEXCOMM5.

enumerator kFRO_HF_to_FLEXCOMM5
Attach FRO_HF to FLEXCOMM5.

enumerator kSYS_PLL_to_FLEXCOMM5
Attach SYS_PLL to FLEXCOMM5.

enumerator kMCLK_to_FLEXCOMM5
Attach MCLK to FLEXCOMM5.

enumerator kFRG_to_FLEXCOMM5
Attach FRG to FLEXCOMM5.

enumerator kNONE_to_FLEXCOMM5
Attach NONE to FLEXCOMM5.

enumerator kFRO12M_to_FLEXCOMM6
Attach FRO12M to FLEXCOMM6.

enumerator kFRO_HF_to_FLEXCOMM6
Attach FRO_HF to FLEXCOMM6.

enumerator kSYS_PLL_to_FLEXCOMM6
Attach SYS_PLL to FLEXCOMM6.

enumerator kMCLK_to_FLEXCOMM6
Attach MCLK to FLEXCOMM6.

enumerator kFRG_to_FLEXCOMM6
Attach FRG to FLEXCOMM6.

enumerator kNONE_to_FLEXCOMM6
Attach NONE to FLEXCOMM6.

enumerator kFRO12M_to_FLEXCOMM7
Attach FRO12M to FLEXCOMM7.

enumerator kFRO_HF_to_FLEXCOMM7
Attach FRO_HF to FLEXCOMM7.

enumerator kSYS_PLL_to_FLEXCOMM7
Attach SYS_PLL to FLEXCOMM7.

enumerator kMCLK_to_FLEXCOMM7
Attach MCLK to FLEXCOMM7.

enumerator kFRG_to_FLEXCOMM7
Attach FRG to FLEXCOMM7.

enumerator kNONE_to_FLEXCOMM7
Attach NONE to FLEXCOMM7.

enumerator kMAIN_CLK_to_FRG
Attach MAIN_CLK to FRG.

enumerator kSYS_PLL_to_FRG
Attach SYS_PLL to FRG.

enumerator kFRO12M_to_FRG
Attach FRO12M to FRG.

enumerator kFRO_HF_to_FRG
Attach FRO_HF to FRG.

enumerator kNONE_to_FRG
Attach NONE to FRG.

enumerator kFRO_HF_to_MCLK
Attach FRO_HF to MCLK.

enumerator kSYS_PLL_to_MCLK
Attach SYS_PLL to MCLK.

enumerator kMAIN_CLK_to_MCLK
Attach MAIN_CLK to MCLK.

enumerator kNONE_to_MCLK
Attach NONE to MCLK.

enumerator kFRO_HF_to_USB_CLK
Attach FRO_HF to USB_CLK.

enumerator kSYS_PLL_to_USB_CLK
Attach SYS_PLL to USB_CLK.

enumerator kMAIN_CLK_to_USB_CLK
Attach MAIN_CLK to USB_CLK.

enumerator kNONE_to_USB_CLK
Attach NONE to USB_CLK.

enumerator kMAIN_CLK_to_CLKOUT
Attach MAIN_CLK to CLKOUT.

enumerator kEXT_CLK_to_CLKOUT
Attach EXT_CLK to CLKOUT.

enumerator kWDT_OSC_to_CLKOUT
Attach WDT_OSC to CLKOUT.

enumerator kFRO_HF_to_CLKOUT
Attach FRO_HF to CLKOUT.

enumerator kSYS_PLL_to_CLKOUT
Attach SYS_PLL to CLKOUT.

enumerator kFRO12M_to_CLKOUT
Attach FRO12M to CLKOUT.

enumerator kOSC32K_to_CLKOUT
Attach OSC32K to CLKOUT.

enumerator kNONE_to_CLKOUT
Attach NONE to CLKOUT.

enumerator kNONE_to_NONE
Attach NONE to NONE.

enum _clock_div_name
Clock dividers.

Values:

enumerator kCLOCK_DivSystickClk
Systick clock divider.

enumerator kCLOCK_DivTraceClk
Trace clock divider.

enumerator kCLOCK_DivAhbClk
Ahb clock divider.

enumerator kCLOCK_DivClkOut
Clock out divider.

enumerator kCLOCK_DivAdcAsyncClk
Adc Async clock divider.

enumerator kCLOCK_DivUsbClk
Usb clock divider.

enumerator kCLOCK_DivFrg
Frg clock divider.

enumerator kCLOCK_DivFxi2s0MClk
Fxi2s0 clock divider.

enum _clock_flashtim
FLASH Access time definitions.

Values:

enumerator kCLOCK_Flash1Cycle
Flash accesses use 1 CPU clock

enumerator kCLOCK_Flash2Cycle
Flash accesses use 2 CPU clocks

enumerator kCLOCK_Flash3Cycle
Flash accesses use 3 CPU clocks

enumerator kCLOCK_Flash4Cycle
Flash accesses use 4 CPU clocks

enumerator kCLOCK_Flash5Cycle
Flash accesses use 5 CPU clocks

enumerator kCLOCK_Flash6Cycle
Flash accesses use 6 CPU clocks

enumerator kCLOCK_Flash7Cycle
Flash accesses use 7 CPU clocks

enum _ss_progmodfm

PLL Spread Spectrum (SS) Programmable modulation frequency See (MF) field in the SYS-PLLSSCTRL1 register in the UM.

Values:

enumerator kSS_MF_512
Nss = 512 (fm ? 3.9 - 7.8 kHz)

enumerator kSS_MF_384
Nss ?= 384 (fm ? 5.2 - 10.4 kHz)

enumerator kSS_MF_256
Nss = 256 (fm ? 7.8 - 15.6 kHz)

enumerator kSS_MF_128
Nss = 128 (fm ? 15.6 - 31.3 kHz)

enumerator kSS_MF_64
Nss = 64 (fm ? 32.3 - 64.5 kHz)

enumerator kSS_MF_32
Nss = 32 (fm ? 62.5- 125 kHz)

enumerator kSS_MF_24
Nss ?= 24 (fm ? 83.3- 166.6 kHz)

enumerator kSS_MF_16
Nss = 16 (fm ? 125- 250 kHz)

enum _ss_progmoddp

PLL Spread Spectrum (SS) Programmable frequency modulation depth See (MR) field in the SYSPLLSSCTRL1 register in the UM.

Values:

enumerator kSS_MR_K0
k = 0 (no spread spectrum)

enumerator kSS_MR_K1
k = 1

enumerator kSS_MR_K1_5
k = 1.5

enumerator kSS_MR_K2

k = 2

enumerator kSS_MR_K3

k = 3

enumerator kSS_MR_K4

k = 4

enumerator kSS_MR_K6

k = 6

enumerator kSS_MR_K8

k = 8

enum _ss_modwvctrl

PLL Spread Spectrum (SS) Modulation waveform control See (MC) field in the SYSPLLSSC-TRL1 register in the UM.

Compensation for low pass filtering of the PLL to get a triangular modulation at the output of the PLL, giving a flat frequency spectrum.

Values:

enumerator kSS_MC_NOC

no compensation

enumerator kSS_MC_RECC

recommended setting

enumerator kSS_MC_MAXC

max. compensation

enum _pll_error

PLL status definitions.

Values:

enumerator kStatus_PLL_Success

PLL operation was successful

enumerator kStatus_PLL_OutputTooLow

PLL output rate request was too low

enumerator kStatus_PLL_OutputTooHigh

PLL output rate request was too high

enumerator kStatus_PLL_InputTooLow

PLL input rate is too low

enumerator kStatus_PLL_InputTooHigh

PLL input rate is too high

enumerator kStatus_PLL_OutsideIntLimit

Requested output rate isn't possible

enum _clock_usb_src

USB clock source definition.

Values:

enumerator kCLOCK_UsbSrcFro

Use FRO 96 or 48 MHz.

enumerator `kCLOCK__UsbSrcSystemPll`
Use System PLL output.

enumerator `kCLOCK__UsbSrcMainClock`
Use Main clock.

enumerator `kCLOCK__UsbSrcNone`
Use None, this may be selected in order to reduce power when no output is needed.

typedef enum `_clock_ip_name` `clock_ip_name_t`
Clock gate name used for `CLOCK_EnableClock/CLOCK_DisableClock`.

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

typedef enum `_async_clock_src` `async_clock_src_t`
Clock source selections for the asynchronous APB clock.

typedef enum `_clock_attach_id` `clock_attach_id_t`
The enumerator of clock attach Id.

typedef enum `_clock_div_name` `clock_div_name_t`
Clock dividers.

typedef enum `_clock_flashtim` `clock_flashtim_t`
FLASH Access time definitions.

typedef enum `_ss_progmodfm` `ss_progmodfm_t`
PLL Spread Spectrum (SS) Programmable modulation frequency See (MF) field in the `SYSPLLSSCTRL1` register in the UM.

typedef enum `_ss_progmoddp` `ss_progmoddp_t`
PLL Spread Spectrum (SS) Programmable frequency modulation depth See (MR) field in the `SYSPLLSSCTRL1` register in the UM.

typedef enum `_ss_modwvctrl` `ss_modwvctrl_t`
PLL Spread Spectrum (SS) Modulation waveform control See (MC) field in the `SYSPLLSSCTRL1` register in the UM.
Compensation for low pass filtering of the PLL to get a triangular modulation at the output of the PLL, giving a flat frequency spectrum.

typedef struct `_pll_config` `pll_config_t`
PLL configuration structure.
This structure can be used to configure the settings for a PLL setup structure. Fill in the desired configuration for the PLL and call the PLL setup function to fill in a PLL setup structure.

typedef struct `_pll_setup` `pll_setup_t`
PLL setup structure This structure can be used to pre-build a PLL setup configuration at run-time and quickly set the PLL to the configuration. It can be populated with the PLL setup function. If powering up or waiting for PLL lock, the PLL input clock source should be configured prior to PLL setup.

typedef enum `_pll_error` `pll_error_t`
PLL status definitions.

typedef enum `_clock_usb_src` `clock_usb_src_t`
USB clock source definition.

static inline void `CLOCK_EnableClock(clock_ip_name_t clk)`

static inline void CLOCK_DisableClock(*clock_ip_name_t* clk)

static inline void CLOCK_SetFLASHAccessCycles(*clock_flashtim_t* clks)

Set FLASH memory access time in clocks.

Parameters

- clks – : Clock cycles for FLASH access

Returns

Nothing

status_t CLOCK_SetupFROClocking(uint32_t iFreq)

Initialize the Core clock to given frequency (12, 48 or 96 MHz). Turns on FRO and uses default CCO, if freq is 12000000, then high speed output is off, else high speed output is enabled.

Parameters

- iFreq – : Desired frequency (must be one of CLK_FRO_12MHZ or CLK_FRO_48MHZ or CLK_FRO_96MHZ)

Returns

returns success or fail status.

void CLOCK_AttachClk(*clock_attach_id_t* connection)

Configure the clock selection muxes.

Parameters

- connection – : Clock to be configured.

Returns

Nothing

clock_attach_id_t CLOCK_GetClockAttachId(*clock_attach_id_t* attachId)

Get the actual clock attach id. This function uses the offset in input attach id, then it reads the actual source value in the register and combine the offset to obtain an actual attach id.

Parameters

- attachId – : Clock attach id to get.

Returns

Clock source value.

void CLOCK_SetClkDiv(*clock_div_name_t* div_name, uint32_t divided_by_value, bool reset)

Setup peripheral clock dividers.

Parameters

- div_name – : Clock divider name
- divided_by_value – Value to be divided
- reset – : Whether to reset the divider counter.

Returns

Nothing

void CLOCK_SetFLASHAccessCyclesForFreq(uint32_t iFreq)

Set the flash wait states for the input frequency.

Parameters

- iFreq – : Input frequency

Returns

Nothing

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Return Frequency of selected clock.

Returns

Frequency of selected clock

uint32_t CLOCK_GetFRGInputClock(void)

Return Input frequency for the Fractional baud rate generator.

Returns

Input Frequency for FRG

uint32_t CLOCK_SetFRGClock(uint32_t freq)

Set output of the Fractional baud rate generator.

Parameters

- freq – : Desired output frequency

Returns

Error Code 0 - fail 1 - success

uint32_t CLOCK_GetFro12MFreq(void)

Return Frequency of FRO 12MHz.

Returns

Frequency of FRO 12MHz

uint32_t CLOCK_GetExtClkFreq(void)

Return Frequency of External Clock.

Returns

Frequency of External Clock. If no external clock is used returns 0.

uint32_t CLOCK_GetWdtOscFreq(void)

Return Frequency of Watchdog Oscillator.

Returns

Frequency of Watchdog Oscillator

uint32_t CLOCK_GetFroHfFreq(void)

Return Frequency of High-Freq output of FRO.

Returns

Frequency of High-Freq output of FRO

uint32_t CLOCK_GetPllOutFreq(void)

Return Frequency of PLL.

Returns

Frequency of PLL

uint32_t CLOCK_GetOsc32KFreq(void)

Return Frequency of 32kHz osc.

Returns

Frequency of 32kHz osc

uint32_t CLOCK_GetCoreSysClkFreq(void)

Return Frequency of Core System.

Returns

Frequency of Core System

uint32_t CLOCK_GetI2SMClkFreq(void)

Return Frequency of I2S MCLK Clock.

Returns

Frequency of I2S MCLK Clock

uint32_t CLOCK_GetFlexCommClkFreq(uint32_t id)

Return Frequency of Flexcomm functional Clock.

Returns

Frequency of Flexcomm functional Clock

uint32_t CLOCK_GetUsbClkFreq(void)

brief Return Frequency of Usb Clock return Frequency of Usb Clock.

uint32_t CLOCK_GetAdcClkFreq(void)

Return Frequency of Adc Clock.

Returns

Frequency of Adc Clock.

uint32_t CLOCK_GetClockOutClkFreq(void)

Return Frequency of ClockOut.

Returns

Frequency of ClockOut

__STATIC_INLINE async_clock_src_t CLOCK_GetAsyncApbClkSrc (void)

Return Asynchronous APB Clock source.

Returns

Asynchronous APB CLock source

uint32_t CLOCK_GetAsyncApbClkFreq(void)

Return Frequency of Asynchronous APB Clock.

Returns

Frequency of Asynchronous APB Clock Clock

uint32_t CLOCK_GetSystemPLLInClockRate(void)

Return System PLL input clock rate.

Returns

System PLL input clock rate

uint32_t CLOCK_GetSystemPLLOutClockRate(bool recompute)

Return System PLL output clock rate.

Note: The PLL rate is cached in the driver in a variable as the rate computation function can take some time to perform. It is recommended to use 'false' with the 'recompute' parameter.

Parameters

- recompute – : Forces a PLL rate recomputation if true

Returns

System PLL output clock rate

__STATIC_INLINE void CLOCK_SetBypassPLL (bool bypass)

Enables and disables PLL bypass mode.

bypass : true to bypass PLL (PLL output = PLL input, false to disable bypass)

Returns

System PLL output clock rate

__STATIC_INLINE bool CLOCK_IsSystemPLLLocked (void)

Check if PLL is locked or not.

Returns

true if the PLL is locked, false if not locked

void CLOCK_SetStoredPLLClockRate(uint32_t rate)

Store the current PLL rate.

Parameters

- rate – Current rate of the PLL

Returns

Nothing

uint32_t CLOCK_GetSystemPLLOutFromSetup(*pll_setup_t* *pSetup)

Return System PLL output clock rate from setup structure.

Parameters

- pSetup – : Pointer to a PLL setup structure

Returns

System PLL output clock rate calculated from the setup structure

pll_error_t CLOCK_SetupPLLData(*pll_config_t* *pControl, *pll_setup_t* *pSetup)

Set PLL output based on the passed PLL setup data.

Note: Actual frequency for setup may vary from the desired frequency based on the accuracy of input clocks, rounding, non-fractional PLL mode, etc.

Parameters

- pControl – : Pointer to populated PLL control structure to generate setup with
- pSetup – : Pointer to PLL setup structure to be filled

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

pll_error_t CLOCK_SetupSystemPLLPrec(*pll_setup_t* *pSetup, uint32_t flagcfg)

Set PLL output from PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- pSetup – : Pointer to populated PLL setup structure
- flagcfg – : Flag configuration for PLL config structure

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

pll_error_t CLOCK_SetPLLFreq(const *pll_setup_t* *pSetup)

Set PLL output from PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- pSetup – : Pointer to populated PLL setup structure

Returns

kStatus_PLL_Success on success, or PLL setup error code

void CLOCK_SetupSystemPLLMult(uint32_t multiply_by, uint32_t input_freq)

Set PLL output based on the multiplier and input frequency.

Note: Unlike the Chip_Clock_SetupSystemPLLPrec() function, this function does not disable or enable PLL power, wait for PLL lock, or adjust system voltages. These must be done in the application. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- multiply_by – : multiplier
- input_freq – : Clock input frequency of the PLL

Returns

Nothing

static inline void CLOCK_DisableUsbfs0Clock(void)

Disable USB FS clock.

Disable USB FS clock.

bool CLOCK_EnableUsbfs0Clock(*clock_usb_src_t* src, uint32_t freq)

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.4.2.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

CLOCK_USR_CFG_PLL_CONFIG_CACHE_COUNT

User-defined the size of cache for CLOCK_PllGetConfig() function.

Once define this MACRO to be non-zero value, CLOCK_PllGetConfig() function would cache the recent calulation and accelerate the execution to get the right settings.

FLEXCOMM_CLOCKS

Clock ip name array for FLEXCOMM.

LPUART_CLOCKS

Clock ip name array for LPUART.

BI2C_CLOCKS

Clock ip name array for BI2C.

LPSI_CLOCKS

Clock ip name array for LSPI.

FLEXI2S_CLOCKS

Clock ip name array for FLEXI2S.

UTICK_CLOCKS

Clock ip name array for UTICK.

DMA_CLOCKS

Clock ip name array for DMA.

CTIMER_CLOCKS

Clock ip name array for CT32B.

GPIO_CLOCKS

Clock ip name array for GPIO.

ADC_CLOCKS

Clock ip name array for ADC.

MRT_CLOCKS

Clock ip name array for MRT.

SCT_CLOCKS

Clock ip name array for MRT.

RTC_CLOCKS

Clock ip name array for RTC.

WWDT_CLOCKS

Clock ip name array for WWDT.

CRC_CLOCKS

Clock ip name array for CRC.

USBD_CLOCKS

Clock ip name array for USBD.

GINT_CLOCKS

Clock ip name array for GINT. GINT0 & GINT1 share same slot.

CLK_GATE_REG_OFFSET_SHIFT

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

CLK_GATE_REG_OFFSET_MASK

CLK_GATE_BIT_SHIFT_SHIFT

CLK_GATE_BIT_SHIFT_MASK

CLK_GATE_DEFINE(reg_offset, bit_shift)

CLK_GATE_ABSTRACT_REG_OFFSET(x)

CLK_GATE_ABSTRACT_BITS_SHIFT(x)

AHB_CLK_CTRL0

AHB_CLK_CTRL1

ASYNC_CLK_CTRL0

CLK_ATTACH_ID(mux, sel, pos)

Clock Mux Switches The encoding is as follows each connection identified is 32bits wide while 24bits are valuable starting from LSB upwards.

[4 bits for choice, 0 means invalid choice] [8 bits mux ID]*

MUX_A(mux, sel)
MUX_B(mux, sel, selector)
GET_ID_ITEM(connection)
GET_ID_NEXT_ITEM(connection)
GET_ID_ITEM_MUX(connection)
GET_ID_ITEM_SEL(connection)
GET_ID_SELECTOR(connection)
CM_MAINCLKSELA
CM_MAINCLKSELB
CM_CLKOUTCLKSELA
CM_CLKOUTCLKSELB
CM_SYSPLLCLKSEL
CM_USBPPLLCLKSEL
CM_AUDPLLCLKSEL
CM_SCTPLLCLKSEL
CM_ADCASYNCCLKSEL
CM_USBCLKSEL
CM_USB1CLKSEL
CM_FXCOMCLKSEL0
CM_FXCOMCLKSEL1
CM_FXCOMCLKSEL2
CM_FXCOMCLKSEL3
CM_FXCOMCLKSEL4
CM_FXCOMCLKSEL5
CM_FXCOMCLKSEL6
CM_FXCOMCLKSEL7
CM_FXCOMCLKSEL8
CM_FXCOMCLKSEL9
CM_FXCOMCLKSEL10
CM_FXCOMCLKSEL11
CM_FXI2S0MCLKCLKSEL
CM_FXI2S1MCLKCLKSEL
CM_FRGCLKSEL

CM_ASYNCAPB

PLL_CONFIGFLAG_USEINRATE

PLL configuration structure flags for 'flags' field These flags control how the PLL configuration function sets up the PLL setup structure.

When the PLL_CONFIGFLAG_USEINRATE flag is selected, the 'InputRate' field in the configuration structure must be assigned with the expected PLL frequency. If the PLL_CONFIGFLAG_USEINRATE is not used, 'InputRate' is ignored in the configuration function and the driver will determine the PLL rate from the currently selected PLL source. This flag might be used to configure the PLL input clock more accurately when using the WDT oscillator or a more dynamic CLKIN source.

When the PLL_CONFIGFLAG_FORCENOFRACT flag is selected, the PLL hardware for the automatic bandwidth selection, Spread Spectrum (SS) support, and fractional M-divider are not used.

Flag to use InputRate in PLL configuration structure for setup

PLL_CONFIGFLAG_FORCENOFRACT

Force non-fractional output mode, PLL output will not use the fractional, automatic bandwidth, or \ SS hardware

PLL_SETUPFLAG_POWERUP

PLL setup structure flags for 'flags' field These flags control how the PLL setup function sets up the PLL.

Setup will power on the PLL after setup

PLL_SETUPFLAG_WAITLOCK

Setup will wait for PLL lock, implies the PLL will be powered on

PLL_SETUPFLAG_ADGVOLT

Optimize system voltage for the new PLL rate

PLL_SETUPFLAG_USEFEEDBACKDIV2

Use feedback divider by 2 in divider path

uint32_t desiredRate

Desired PLL rate in Hz

uint32_t inputRate

PLL input clock in Hz, only used if PLL_CONFIGFLAG_USEINRATE flag is set

uint32_t flags

PLL configuration flags, Or'ed value of PLL_CONFIGFLAG_* definitions

ss_progmodfm_t ss_mf

SS Programmable modulation frequency, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

ss_progmoddp_t ss_mr

SS Programmable frequency modulation depth, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

ss_modwvctrl_t ss_mc

SS Modulation waveform control, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

bool mfDither

false for fixed modulation frequency or true for dithering, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

uint32_t syspllctrl

PLL control register SYSPLLCTRL

uint32_t syspllndec

PLL NDEC register SYSPLLNDEC

uint32_t syspllpdec

PLL PDEC register SYSPLLPDEC

uint32_t syspllssctrl[2]

PLL SSCTL registers SYSPLLSSCTRL

uint32_t pllRate

Actual PLL rate

uint32_t flags

PLL setup flags, Or'ed value of PLL_SETUPFLAG_* definitions

struct __pll_config

#include <fsl_clock.h> PLL configuration structure.

This structure can be used to configure the settings for a PLL setup structure. Fill in the desired configuration for the PLL and call the PLL setup function to fill in a PLL setup structure.

struct __pll_setup

#include <fsl_clock.h> PLL setup structure This structure can be used to pre-build a PLL setup configuration at run-time and quickly set the PLL to the configuration. It can be populated with the PLL setup function. If powering up or waiting for PLL lock, the PLL input clock source should be configured prior to PLL setup.

2.2 CRC: Cyclic Redundancy Check Driver

FSL_CRC_DRIVER_VERSION

CRC driver version. Version 2.1.1.

Current version: 2.1.1

Change log:

- Version 2.0.0
 - initial version
- Version 2.0.1
 - add explicit type cast when writing to WR_DATA
- Version 2.0.2
 - Fix MISRA issue
- Version 2.1.0
 - Add CRC_WriteSeed function
- Version 2.1.1
 - Fix MISRA issue

enum __crc_polynomial

CRC polynomials to use.

Values:

enumerator kCRC_Polynomial_CRC_CCITT
 $x^{16}+x^{12}+x^5+1$

enumerator kCRC_Polynomial_CRC_16
 $x^{16}+x^{15}+x^2+1$

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

typedef enum *_crc_polynomial* crc_polynomial_t
CRC polynomials to use.

typedef struct *_crc_config* crc_config_t
CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

void CRC_Init(CRC_Type *base, const *crc_config_t* *config)
Enables and configures the CRC peripheral module.

This functions enables the CRC peripheral clock in the LPC SYSCON block. It also configures the CRC engine and starts checksum computation by writing the seed.

Parameters

- base – CRC peripheral address.
- config – CRC module configuration structure.

static inline void CRC_Deinit(CRC_Type *base)
Disables the CRC peripheral module.

This functions disables the CRC peripheral clock in the LPC SYSCON block.

Parameters

- base – CRC peripheral address.

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

Parameters

- base – CRC peripheral address.

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

Parameters

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

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

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

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

Parameters

- `config` – CRC protocol configuration structure

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

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

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

Parameters

- `base` – CRC peripheral address.
- `config` – CRC protocol configuration structure

`void CRC_WriteData(CRC_Type *base, const uint8_t *data, size_t dataSize)`

Writes data to the CRC module.

Writes input data buffer bytes to CRC data register.

Parameters

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

`static inline uint32_t CRC_Get32bitResult(CRC_Type *base)`

Reads 32-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

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

`static inline uint16_t CRC_Get16bitResult(CRC_Type *base)`

Reads 16-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

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

`CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT`

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

`struct _crc_config`

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

This structure holds the configuration for the CRC protocol.

Public Members

`crc_polynomial_t` polynomial

CRC polynomial.

`bool` reverseIn

Reverse bits on input.

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

`bool reverseOut`
Reverse bits on output.

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

`uint32_t seed`
Starting checksum value.

2.3 CTIMER: Standard counter/timers

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

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

Parameters

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

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

Parameters

- `base` – Ctimer peripheral base address

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

The default values are:

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

Parameters

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

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

Configures the PWM signal parameters.

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

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

Parameters

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

Returns

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

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

Configures the PWM signal parameters.

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

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

Parameters

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

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

Updates the pulse period of an active PWM signal.

Parameters

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

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

Updates the duty cycle of an active PWM signal.

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

Parameters

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

Returns

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

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

Enables the selected Timer interrupts.

Parameters

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

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

Disables the selected Timer interrupts.

Parameters

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

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

Gets the enabled Timer interrupts.

Parameters

- `base` – Ctimer peripheral base address

Returns

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

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

Gets the Timer status flags.

Parameters

- `base` – Ctimer peripheral base address

Returns

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

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

Clears the Timer status flags.

Parameters

- `base` – Ctimer peripheral base address

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

static inline void CTIMER_StartTimer(CTIMER_Type *base)

Starts the Timer counter.

Parameters

- base – Ctimer peripheral base address

static inline void CTIMER_StopTimer(CTIMER_Type *base)

Stops the Timer counter.

Parameters

- base – Ctimer peripheral base address

FSL_CTIMER_DRIVER_VERSION

Version 2.3.3

enum _ctimer_capture_channel

List of Timer capture channels.

Values:

enumerator kCTIMER_Capture_0

Timer capture channel 0

enumerator kCTIMER_Capture_1

Timer capture channel 1

enumerator kCTIMER_Capture_3

Timer capture channel 3

enum _ctimer_capture_edge

List of capture edge options.

Values:

enumerator kCTIMER_Capture_RiseEdge

Capture on rising edge

enumerator kCTIMER_Capture_FallEdge

Capture on falling edge

enumerator kCTIMER_Capture_BothEdge

Capture on rising and falling edge

enum _ctimer_match

List of Timer match registers.

Values:

enumerator kCTIMER_Match_0

Timer match register 0

enumerator kCTIMER_Match_1

Timer match register 1

enumerator kCTIMER_Match_2

Timer match register 2

enumerator kCTIMER_Match_3

Timer match register 3

enum `_ctimer_external_match`

List of external match.

Values:

enumerator `kCTIMER_External_Match_0`
External match 0

enumerator `kCTIMER_External_Match_1`
External match 1

enumerator `kCTIMER_External_Match_2`
External match 2

enumerator `kCTIMER_External_Match_3`
External match 3

enum `_ctimer_match_output_control`

List of output control options.

Values:

enumerator `kCTIMER_Output_NoAction`
No action is taken

enumerator `kCTIMER_Output_Clear`
Clear the EM bit/output to 0

enumerator `kCTIMER_Output_Set`
Set the EM bit/output to 1

enumerator `kCTIMER_Output_Toggle`
Toggle the EM bit/output

enum `_ctimer_timer_mode`

List of Timer modes.

Values:

enumerator `kCTIMER_TimerMode`

enumerator `kCTIMER_IncreaseOnRiseEdge`

enumerator `kCTIMER_IncreaseOnFallEdge`

enumerator `kCTIMER_IncreaseOnBothEdge`

enum `_ctimer_interrupt_enable`

List of Timer interrupts.

Values:

enumerator `kCTIMER_Match0InterruptEnable`
Match 0 interrupt

enumerator `kCTIMER_Match1InterruptEnable`
Match 1 interrupt

enumerator `kCTIMER_Match2InterruptEnable`
Match 2 interrupt

enumerator `kCTIMER_Match3InterruptEnable`
Match 3 interrupt

enum `_ctimer_status_flags`

List of Timer flags.

Values:

enumerator `kCTIMER_Match0Flag`

Match 0 interrupt flag

enumerator `kCTIMER_Match1Flag`

Match 1 interrupt flag

enumerator `kCTIMER_Match2Flag`

Match 2 interrupt flag

enumerator `kCTIMER_Match3Flag`

Match 3 interrupt flag

enum `ctimer_callback_type_t`

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

Values:

enumerator `kCTIMER_SingleCallback`

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

enumerator `kCTIMER_MultipleCallback`

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

typedef enum `_ctimer_capture_channel` `ctimer_capture_channel_t`

List of Timer capture channels.

typedef enum `_ctimer_capture_edge` `ctimer_capture_edge_t`

List of capture edge options.

typedef enum `_ctimer_match` `ctimer_match_t`

List of Timer match registers.

typedef enum `_ctimer_external_match` `ctimer_external_match_t`

List of external match.

typedef enum `_ctimer_match_output_control` `ctimer_match_output_control_t`

List of output control options.

typedef enum `_ctimer_timer_mode` `ctimer_timer_mode_t`

List of Timer modes.

typedef enum `_ctimer_interrupt_enable` `ctimer_interrupt_enable_t`

List of Timer interrupts.

typedef enum `_ctimer_status_flags` `ctimer_status_flags_t`

List of Timer flags.

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

typedef struct `_ctimer_match_config` `ctimer_match_config_t`

Match configuration.

This structure holds the configuration settings for each match register.

```
typedef struct _ctimer_config ctimer_config_t
```

Timer configuration structure.

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

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

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

Setup the match register.

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

Parameters

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

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

Get the status of output match.

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

Parameters

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

Returns

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

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

Setup the capture.

Parameters

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

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

Get the timer count value from TC register.

Parameters

- base – Ctimer peripheral base address.

Returns

return the timer count value.

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

Register callback.

This function configures CTimer Callback in following modes:

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

Parameters

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

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

Reset the counter.

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

Parameters

- *base* – Ctimer peripheral base address

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

Setup the timer prescale value.

Specifies the maximum value for the Prescale Counter.

Parameters

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

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

Get capture channel value.

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

Parameters

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

Returns

The timer count capture value.

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

Enable reset match channel.

Set the specified match channel reset operation.

Parameters

- *base* – Ctimer peripheral base address

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

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

Enable stop match channel.

Set the specified match channel stop operation.

Parameters

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

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

Enable reload channel falling edge.

Enable the specified match channel reload match shadow value.

Parameters

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

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

Enable capture channel rising edge.

Sets the specified capture channel for rising edge capture.

Parameters

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

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

Enable capture channel falling edge.

Sets the specified capture channel for falling edge capture.

Parameters

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

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

Set the specified match shadow channel.

Parameters

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

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

`struct _ctimer_match_config`

#include <fsl_ctimer.h> Match configuration.

This structure holds the configuration settings for each match register.

Public Members

`uint32_t matchValue`

This is stored in the match register

`bool enableCounterReset`

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

`bool enableCounterStop`

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

`ctimer_match_output_control_t outControl`

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

`bool outPinInitState`

Initial value of the EM bit/output

`bool enableInterrupt`

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

`struct _ctimer_config`

#include <fsl_ctimer.h> Timer configuration structure.

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

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

Public Members

`ctimer_timer_mode_t mode`

Timer mode

`ctimer_capture_channel_t input`

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

`uint32_t prescale`

Prescale value

2.4 DMA: Direct Memory Access Controller Driver

`void DMA_Init(DMA_Type *base)`

Initializes DMA peripheral.

This function enables the DMA clock, sets descriptor table and enables DMA peripheral.

Parameters

- `base` – DMA peripheral base address.

`void DMA_Deinit(DMA_Type *base)`

Deinitializes DMA peripheral.

This function gates the DMA clock.

Parameters

- `base` – DMA peripheral base address.

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

Install DMA descriptor memory.

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

Parameters

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

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

Return whether DMA channel is processing transfer.

Parameters

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

Returns

True for active state, false otherwise.

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

Return whether DMA channel is busy.

Parameters

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

Returns

True for busy state, false otherwise.

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

Enables the interrupt source for the DMA transfer.

Parameters

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

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

Disables the interrupt source for the DMA transfer.

Parameters

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

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

Enable DMA channel.

Parameters

- `base` – DMA peripheral base address.

- channel – DMA channel number.

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

Disable DMA channel.

Parameters

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

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

Set PERIPHREQEN of channel configuration register.

Parameters

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

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

Get PERIPHREQEN value of channel configuration register.

Parameters

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

Returns

True for enabled PeriphRq, false for disabled.

void DMA_ConfigureChannelTrigger(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger)

Set trigger settings of DMA channel.

Deprecated:

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

Parameters

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

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

set channel config.

This function provide a interface to configure channel configuration registers.

Parameters

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

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

DMA channel xfer transfer configurations.

Parameters

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

Returns

The vaule of xfer config

uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)

Gets the remaining bytes of the current DMA descriptor transfer.

Parameters

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

Returns

The number of bytes which have not been transferred yet.

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

Set priority of channel configuration register.

Parameters

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

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

Get priority of channel configuration register.

Parameters

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

Returns

Channel priority value.

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

Set channel configuration valid.

Parameters

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

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

Do software trigger for the channel.

Parameters

- base – DMA peripheral base address.

- channel – DMA channel number.

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

Load channel transfer configurations.

Parameters

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

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

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

Deprecated:

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

Parameters

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

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

setup dma descriptor

Note: This function do not support configure wrap descriptor.

Parameters

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

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

setup dma channel descriptor

Note: This function support configure wrap descriptor.

Parameters

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

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

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

load channel transfer decriptor.

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

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

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

Parameters

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

`void DMA_AbortTransfer(dma_handle_t *handle)`

Abort running transfer by handle.

This function aborts DMA transfer specified by handle.

Parameters

- handle – DMA handle pointer.

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

Creates the DMA handle.

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

Parameters

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

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

Installs a callback function for the DMA transfer.

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

Parameters

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

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

Prepares the DMA transfer structure.

Deprecated:

Do not use this function. It has been superceded by DMA_PrepareChannelTransfer. This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, so the source address must be 4 bytes aligned, or it shall result in source address error(SAE).

Parameters

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

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

Prepare channel transfer configurations.

This function used to prepare channel transfer configurations.

Parameters

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

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

Submits the DMA transfer request.

Deprecated:

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

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

Parameters

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

Return values

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

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

Submit channel transfer paramter directly.

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

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

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

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

```
define link descriptor table in application with macro  
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[3]);  
  
DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,  
↪ srcInc, dstInc, bytes),  
srcStartAddr, dstStartAddr, nextDesc1);  
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,  
↪ srcInc, dstInc, bytes),  
srcStartAddr, dstStartAddr, nextDesc2);  
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,  
↪ srcInc, dstInc, bytes),  
srcStartAddr, dstStartAddr, NULL);  
DMA_SetChannelConfig(base, channel, trigger, isPeriph);  
DMA_CreateHandle(handle, base, channel)  
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,  
↪ intA, intB, width, srcInc, dstInc,  
bytes), srcStartAddr, dstStartAddr, nextDesc0);  
DMA_StartTransfer(handle);
```

Parameters

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

- nextDesc – address of next descriptor.

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

Submit channel descriptor.

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

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

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

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

Parameters

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

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

Submits the DMA channel transfer request.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time. It is used for the case:

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

Parameters

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

Return values

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

void DMA_StartTransfer(*dma_handle_t* *handle)

DMA start transfer.

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

Parameters

- handle – DMA handle pointer.

void DMA_IRQHandle(DMA_Type *base)

DMA IRQ handler for descriptor transfer complete.

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

Parameters

- base – DMA base address.

FSL_DMA_DRIVER_VERSION

DMA driver version.

Version 2.5.3.

_dma_transfer_status DMA transfer status

Values:

enumerator kStatus_DMA_Busy

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

_dma_addr_interleave_size dma address interleave size

Values:

enumerator kDMA_AddressInterleave0xWidth

dma source/destination address no interleave

enumerator kDMA_AddressInterleave1xWidth

dma source/destination address interleave 1xwidth

enumerator kDMA_AddressInterleave2xWidth

dma source/destination address interleave 2xwidth

enumerator kDMA_AddressInterleave4xWidth

dma source/destination address interleave 3xwidth

_dma_transfer_width dma transfer width

Values:

enumerator kDMA_Transfer8BitWidth

dma channel transfer bit width is 8 bit

enumerator kDMA_Transfer16BitWidth

dma channel transfer bit width is 16 bit

enumerator kDMA_Transfer32BitWidth

dma channel transfer bit width is 32 bit

enum _dma_priority

DMA channel priority.

Values:

enumerator kDMA_ChannelPriority0

Highest channel priority - priority 0

enumerator kDMA_ChannelPriority1

Channel priority 1

enumerator kDMA_ChannelPriority2

Channel priority 2

enumerator kDMA_ChannelPriority3

Channel priority 3

enumerator kDMA_ChannelPriority4

Channel priority 4

enumerator kDMA_ChannelPriority5

Channel priority 5

enumerator kDMA_ChannelPriority6

Channel priority 6

enumerator kDMA_ChannelPriority7

Lowest channel priority - priority 7

enum _dma_int

DMA interrupt flags.

Values:

enumerator kDMA_IntA

DMA interrupt flag A

enumerator kDMA_IntB

DMA interrupt flag B

enumerator kDMA_IntError

DMA interrupt flag error

enum _dma_trigger_type

DMA trigger type.

Values:

enumerator kDMA_NoTrigger

Trigger is disabled

enumerator kDMA_LowLevelTrigger

Low level active trigger

enumerator kDMA_HighLevelTrigger

High level active trigger

enumerator kDMA_FallingEdgeTrigger

Falling edge active trigger

enumerator kDMA_RisingEdgeTrigger

Rising edge active trigger

_dma_burst_size DMA burst size

Values:

enumerator kDMA_BurstSize1

burst size 1 transfer

enumerator kDMA_BurstSize2

burst size 2 transfer

enumerator kDMA_BurstSize4

burst size 4 transfer

enumerator kDMA_BurstSize8

burst size 8 transfer

enumerator kDMA_BurstSize16

burst size 16 transfer

enumerator kDMA__BurstSize32
burst size 32 transfer

enumerator kDMA__BurstSize64
burst size 64 transfer

enumerator kDMA__BurstSize128
burst size 128 transfer

enumerator kDMA__BurstSize256
burst size 256 transfer

enumerator kDMA__BurstSize512
burst size 512 transfer

enumerator kDMA__BurstSize1024
burst size 1024 transfer

enum __dma__trigger__burst
DMA trigger burst.

Values:

enumerator kDMA__SingleTransfer
Single transfer

enumerator kDMA__LevelBurstTransfer
Burst transfer driven by level trigger

enumerator kDMA__EdgeBurstTransfer1
Perform 1 transfer by edge trigger

enumerator kDMA__EdgeBurstTransfer2
Perform 2 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer4
Perform 4 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer8
Perform 8 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer16
Perform 16 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer32
Perform 32 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer64
Perform 64 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer128
Perform 128 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer256
Perform 256 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer512
Perform 512 transfers by edge trigger

enumerator kDMA__EdgeBurstTransfer1024
Perform 1024 transfers by edge trigger

enum *_dma_burst_wrap*

DMA burst wrapping.

Values:

enumerator *kDMA_NoWrap*

Wrapping is disabled

enumerator *kDMA_SrcWrap*

Wrapping is enabled for source

enumerator *kDMA_DstWrap*

Wrapping is enabled for destination

enumerator *kDMA_SrcAndDstWrap*

Wrapping is enabled for source and destination

enum *_dma_transfer_type*

DMA transfer type.

Values:

enumerator *kDMA_MemoryToMemory*

Transfer from memory to memory (increment source and destination)

enumerator *kDMA_PeripheralToMemory*

Transfer from peripheral to memory (increment only destination)

enumerator *kDMA_MemoryToPeripheral*

Transfer from memory to peripheral (increment only source)

enumerator *kDMA_StaticToStatic*

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

typedef struct *_dma_descriptor* *dma_descriptor_t*

DMA descriptor structure.

typedef struct *_dma_xfercfg* *dma_xfercfg_t*

DMA transfer configuration.

typedef enum *_dma_priority* *dma_priority_t*

DMA channel priority.

typedef enum *_dma_int* *dma_irq_t*

DMA interrupt flags.

typedef enum *_dma_trigger_type* *dma_trigger_type_t*

DMA trigger type.

typedef enum *_dma_trigger_burst* *dma_trigger_burst_t*

DMA trigger burst.

typedef enum *_dma_burst_wrap* *dma_burst_wrap_t*

DMA burst wrapping.

typedef enum *_dma_transfer_type* *dma_transfer_type_t*

DMA transfer type.

typedef struct *_dma_channel_trigger* *dma_channel_trigger_t*

DMA channel trigger.

typedef struct *_dma_channel_config* *dma_channel_config_t*

DMA channel trigger.

```
typedef struct _dma_transfer_config dma_transfer_config_t
```

DMA transfer configuration.

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

Define Callback function for DMA.

```
typedef struct _dma_handle dma_handle_t
```

DMA transfer handle structure.

```
DMA_MAX_TRANSFER_COUNT
```

DMA max transfer size.

```
FSL_FEATURE_DMA_NUMBER_OF_CHANNELSn(x)
```

DMA channel numbers.

```
FSL_FEATURE_DMA_MAX_CHANNELS
```

```
FSL_FEATURE_DMA_ALL_CHANNELS
```

```
FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE
```

DMA head link descriptor table align size.

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

DMA_ALLOCATE_DATA_TRANSFER_BUFFER(name, width)

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

DMA_CHANNEL_GROUP(channel)

DMA_CHANNEL_INDEX(base, channel)

DMA_COMMON_REG_GET(base, channel, reg)

DMA linked descriptor address align size.

DMA_COMMON_CONST_REG_GET(base, channel, reg)

DMA_COMMON_REG_SET(base, channel, reg, value)

DMA_DESCRIPTOR_END_ADDRESS(start, inc, bytes, width)

DMA descriptor end address calculate.

Parameters

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

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

struct __dma_descriptor

#include <fsl_dma.h> DMA descriptor structure.

Public Members

volatile uint32_t xfercfg

Transfer configuration

void *srcEndAddr

Last source address of DMA transfer

void *dstEndAddr

Last destination address of DMA transfer

void *linkToNextDesc

Address of next DMA descriptor in chain

struct __dma_xfercfg

#include <fsl_dma.h> DMA transfer configuration.

Public Members

bool valid

Descriptor is ready to transfer

bool reload

Reload channel configuration register after current descriptor is exhausted

bool swtrig

Perform software trigger. Transfer if fired when 'valid' is set

bool clrtrig

Clear trigger

`bool intA`
 Raises IRQ when transfer is done and set IRQA status register flag

`bool intB`
 Raises IRQ when transfer is done and set IRQB status register flag

`uint8_t byteWidth`
 Byte width of data to transfer

`uint8_t srcInc`
 Increment source address by 'srcInc' x 'byteWidth'

`uint8_t dstInc`
 Increment destination address by 'dstInc' x 'byteWidth'

`uint16_t transferCount`
 Number of transfers

`struct __dma_channel_trigger`
 #include <fsl_dma.h> DMA channel trigger.

Public Members

`dma_trigger_type_t type`
 Select hardware trigger as edge triggered or level triggered.

`dma_trigger_burst_t burst`
 Select whether hardware triggers cause a single or burst transfer.

`dma_burst_wrap_t wrap`
 Select wrap type, source wrap or dest wrap, or both.

`struct __dma_channel_config`
 #include <fsl_dma.h> DMA channel trigger.

Public Members

`void *srcStartAddr`
 Source data address

`void *dstStartAddr`
 Destination data address

`void *nextDesc`
 Chain custom descriptor

`uint32_t xferCfg`
 channel transfer configurations

`dma_channel_trigger_t *trigger`
 DMA trigger type

`bool isPeriph`
 select the request type

`struct __dma_transfer_config`
 #include <fsl_dma.h> DMA transfer configuration.

Public Members

uint8_t *srcAddr

Source data address

uint8_t *dstAddr

Destination data address

uint8_t *nextDesc

Chain custom descriptor

dma_xfercfg_t xfercfg

Transfer options

bool isPeriph

DMA transfer is driven by peripheral

struct _dma_handle

#include <fsl_dma.h> DMA transfer handle structure.

Public Members

dma_callback callback

Callback function. Invoked when transfer of descriptor with interrupt flag finishes

void *userData

Callback function parameter

DMA_Type *base

DMA peripheral base address

uint8_t channel

DMA channel number

2.5 FLASHIAP: Flash In Application Programming Driver

FSL_FLASHIAP_DRIVER_VERSION

enum _flashiap_status

Flashiap status codes.

Values:

enumerator kStatus_FLASHIAP_Success

Api is executed successfully

enumerator kStatus_FLASHIAP_InvalidCommand

Invalid command

enumerator kStatus_FLASHIAP_SrcAddrError

Source address is not on word boundary

enumerator kStatus_FLASHIAP_DstAddrError

Destination address is not on a correct boundary

enumerator kStatus_FLASHIAP_SrcAddrNotMapped

Source address is not mapped in the memory map

enumerator kStatus_FLASHIAP_DstAddrNotMapped

Destination address is not mapped in the memory map

enumerator kStatus_FLASHIAP_CountError

Byte count is not multiple of 4 or is not a permitted value

enumerator kStatus_FLASHIAP_InvalidSector

Sector number is invalid or end sector number is greater than start sector number

enumerator kStatus_FLASHIAP_SectorNotblank

One or more sectors are not blank

enumerator kStatus_FLASHIAP_NotPrepared

Command to prepare sector for write operation was not executed

enumerator kStatus_FLASHIAP_CompareError

Destination and source memory contents do not match

enumerator kStatus_FLASHIAP_Busy

Flash programming hardware interface is busy

enumerator kStatus_FLASHIAP_ParamError

Insufficient number of parameters or invalid parameter

enumerator kStatus_FLASHIAP_AddrError

Address is not on word boundary

enumerator kStatus_FLASHIAP_AddrNotMapped

Address is not mapped in the memory map

enumerator kStatus_FLASHIAP_NoPower

Flash memory block is powered down

enumerator kStatus_FLASHIAP_NoClock

Flash memory block or controller is not clocked

enum _flashiap_commands

Flashiap command codes.

Values:

enumerator kIapCmd_FLASHIAP_PrepareSectorforWrite

Prepare Sector for write

enumerator kIapCmd_FLASHIAP_CopyRamToFlash

Copy RAM to flash

enumerator kIapCmd_FLASHIAP_EraseSector

Erase Sector

enumerator kIapCmd_FLASHIAP_BlankCheckSector

Blank check sector

enumerator kIapCmd_FLASHIAP_ReadPartId

Read part id

enumerator kIapCmd_FLASHIAP_Read_BootromVersion

Read bootrom version

enumerator kIapCmd_FLASHIAP_Compare

Compare

enumerator kIapCmd_FLASHIAP_ReinvokeISP
Reinvoke ISP

enumerator kIapCmd_FLASHIAP_ReadUid
Read Uid isp

enumerator kIapCmd_FLASHIAP_ErasePage
Erase Page

enumerator kIapCmd_FLASHIAP_ReadMisr
Read Misr

enumerator kIapCmd_FLASHIAP_ReinvokeI2cSpiISP
Reinvoke I2C/SPI isp

typedef void (*FLASHIAP_ENTRY_T)(uint32_t cmd[5], uint32_t stat[4])
IAP_ENTRY API function type.

static inline void iap_entry(uint32_t *cmd_param, uint32_t *status_result)
IAP_ENTRY API function type.

Wrapper for rom iap call

Parameters

- cmd_param – IAP command and relevant parameter array.
- status_result – IAP status result array.

Return values

None. – Status/Result is returned via status_result array.

status_t FLASHIAP_PrepareSectorForWrite(uint32_t startSector, uint32_t endSector)
Prepare sector for write operation.

This function prepares sector(s) for write/erase operation. This function must be called before calling the FLASHIAP_CopyRamToFlash() or FLASHIAP_EraseSector() or FLASHIAP_ErasePage() function. The end sector must be greater than or equal to start sector number.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- startSector – Start sector number.
- endSector – End sector number.

Return values

- kStatus_FLASHIAP_Success – Api was executed successfully.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_InvalidSector – Sector number is invalid or end sector number is greater than start sector number.
- kStatus_FLASHIAP_Busy – Flash programming hardware interface is busy.

```
status_t FLASHIAP_CopyRamToFlash(uint32_t dstAddr, uint32_t *srcAddr, uint32_t  
                                numOfBytes, uint32_t systemCoreClock)
```

Copy RAM to flash.

This function programs the flash memory. Corresponding sectors must be prepared via FLASHIAP_PrepareSectorForWrite before calling this function. The addresses should be a 256 byte boundary and the number of bytes should be 256 | 512 | 1024 | 4096.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- dstAddr – Destination flash address where data bytes are to be written.
- srcAddr – Source ram address from where data bytes are to be read.
- numOfBytes – Number of bytes to be written.
- systemCoreClock – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

- kStatus_FLASHIAP_Success – Api was executed successfully.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_SrcAddrError – Source address is not on word boundary.
- kStatus_FLASHIAP_DstAddrError – Destination address is not on a correct boundary.
- kStatus_FLASHIAP_SrcAddrNotMapped – Source address is not mapped in the memory map.
- kStatus_FLASHIAP_DstAddrNotMapped – Destination address is not mapped in the memory map.
- kStatus_FLASHIAP_CountError – Byte count is not multiple of 4 or is not a permitted value.
- kStatus_FLASHIAP_NotPrepared – Command to prepare sector for write operation was not executed.
- kStatus_FLASHIAP_Busy – Flash programming hardware interface is busy.

```
status_t FLASHIAP_EraseSector(uint32_t startSector, uint32_t endSector, uint32_t  
                              systemCoreClock)
```

Erase sector.

This function erases sector(s). The end sector must be greater than or equal to start sector number. FLASHIAP_PrepareSectorForWrite must be called before calling this function.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- startSector – Start sector number.
- endSector – End sector number.
- systemCoreClock – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

- kStatus_FLASHIAP_Success – Api was executed successfully.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_InvalidSector – Sector number is invalid or end sector number is greater than start sector number.
- kStatus_FLASHIAP_NotPrepared – Command to prepare sector for write operation was not executed.
- kStatus_FLASHIAP_Busy – Flash programming hardware interface is busy.

status_t FLASHIAP_ErasePage(uint32_t startPage, uint32_t endPage, uint32_t systemCoreClock)

This function erases page(s). The end page must be greater than or equal to start page number. Corresponding sectors must be prepared via FLASHIAP_PrepareSectorForWrite before calling this function.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- startPage – Start page number
- endPage – End page number
- systemCoreClock – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

- kStatus_FLASHIAP_Success – Api was executed successfully.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_InvalidSector – Page number is invalid or end page number is greater than start page number
- kStatus_FLASHIAP_NotPrepared – Command to prepare sector for write operation was not executed.
- kStatus_FLASHIAP_Busy – Flash programming hardware interface is busy.

status_t FLASHIAP_BlankCheckSector(uint32_t startSector, uint32_t endSector)

Blank check sector(s)

Blank check single or multiples sectors of flash memory. The end sector must be greater than or equal to start sector number. It can be used to verify the sector erasure after FLASHIAP_EraseSector call.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- startSector – : Start sector number. Must be greater than or equal to start sector number
- endSector – : End sector number

Return values

- kStatus_FLASHIAP_Success – One or more sectors are in erased state.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_SectorNotblank – One or more sectors are not blank.

status_t FLASHIAP_Compare(uint32_t dstAddr, uint32_t *srcAddr, uint32_t numBytes)

Compare memory contents of flash with ram.

This function compares the contents of flash and ram. It can be used to verify the flash memory contents after FLASHIAP_CopyRamToFlash call.

Deprecated:

Do not use this function. It has been moved to iap driver.

Parameters

- dstAddr – Destination flash address.
- srcAddr – Source ram address.
- numBytes – Number of bytes to be compared.

Return values

- kStatus_FLASHIAP_Success – Contents of flash and ram match.
- kStatus_FLASHIAP_NoPower – Flash memory block is powered down.
- kStatus_FLASHIAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_FLASHIAP_AddrError – Address is not on word boundary.
- kStatus_FLASHIAP_AddrNotMapped – Address is not mapped in the memory map.
- kStatus_FLASHIAP_CountError – Byte count is not multiple of 4 or is not a permitted value.
- kStatus_FLASHIAP_CompareError – Destination and source memory contents do not match.

2.6 FLEXCOMM: FLEXCOMM Driver

2.7 FLEXCOMM Driver

FSL_FLEXCOMM_DRIVER_VERSION

FlexCOMM driver version 2.0.2.

enum FLEXCOMM_PERIPH_T

FLEXCOMM peripheral modes.

Values:

enumerator FLEXCOMM_PERIPH_NONE

No peripheral

enumerator FLEXCOMM_PERIPH_USART

USART peripheral

enumerator FLEXCOMM_PERIPH_SPI

SPI Peripheral

enumerator FLEXCOMM_PERIPH_I2C

I2C Peripheral

enumerator FLEXCOMM_PERIPH_I2S_TX

I2S TX Peripheral

enumerator FLEXCOMM_PERIPH_I2S_RX

I2S RX Peripheral

typedef void (*flexcomm_irq_handler_t)(void *base, void *handle)

Typedef for interrupt handler.

IRQn_Type const kFlexcommIrqs[]

Array with IRQ number for each FLEXCOMM module.

uint32_t FLEXCOMM_GetInstance(void *base)

Returns instance number for FLEXCOMM module with given base address.

status_t FLEXCOMM_Init(void *base, FLEXCOMM_PERIPH_T periph)

Initializes FLEXCOMM and selects peripheral mode according to the second parameter.

void FLEXCOMM_SetIRQHandler(void *base, flexcomm_irq_handler_t handler, void *flexcommHandle)

Sets IRQ handler for given FLEXCOMM module. It is used by drivers register IRQ handler according to FLEXCOMM mode.

2.8 FMEAS: Frequency Measure Driver

static inline void FMEAS_StartMeasure(FMEAS_SYSCON_Type *base)

Starts a frequency measurement cycle.

Parameters

- base – : SYSCON peripheral base address.

static inline bool FMEAS_IsMeasureComplete(FMEAS_SYSCON_Type *base)

Indicates when a frequency measurement cycle is complete.

Parameters

- base – : SYSCON peripheral base address.

Returns

true if a measurement cycle is active, otherwise false.

uint32_t FMEAS_GetFrequency(*FMEAS_SYSCON_Type* *base, uint32_t refClockRate)

Returns the computed value for a frequency measurement cycle.

Parameters

- base – : SYSCON peripheral base address.
- refClockRate – : Reference clock rate used during the frequency measurement cycle.

Returns

Frequency in Hz.

FSL_FMEAS_DRIVER_VERSION

Defines LPC Frequency Measure driver version 2.1.1.

typedef SYSCON_Type FMEAS_SYSCON_Type

FMEAS_SYSCON_FREQMECTRL_CAPVAL_MASK

FMEAS_SYSCON_FREQMECTRL_CAPVAL_SHIFT

FMEAS_SYSCON_FREQMECTRL_CAPVAL

FMEAS_SYSCON_FREQMECTRL_PROG_MASK

FMEAS_SYSCON_FREQMECTRL_PROG_SHIFT

FMEAS_SYSCON_FREQMECTRL_PROG

2.9 GINT: Group GPIO Input Interrupt Driver

FSL_GINT_DRIVER_VERSION

Driver version.

enum __gint_comb

GINT combine inputs type.

Values:

enumerator kGINT_CombineOr

A grouped interrupt is generated when any one of the enabled inputs is active

enumerator kGINT_CombineAnd

A grouped interrupt is generated when all enabled inputs are active

enum __gint_trig

GINT trigger type.

Values:

enumerator kGINT_TrigEdge

Edge triggered based on polarity

enumerator kGINT_TrigLevel

Level triggered based on polarity

enum __gint_port

Values:

enumerator kGINT_Port0

enumerator kGINT_Port1

typedef enum *_gint_comb* gint_comb_t

GINT combine inputs type.

typedef enum *_gint_trig* gint_trig_t

GINT trigger type.

typedef enum *_gint_port* gint_port_t

typedef void (*gint_cb_t)(void)

GINT Callback function.

void GINT_Init(GINT_Type *base)

Initialize GINT peripheral.

This function initializes the GINT peripheral and enables the clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

void GINT_SetCtrl(GINT_Type *base, *gint_comb_t* comb, *gint_trig_t* trig, *gint_cb_t* callback)

Setup GINT peripheral control parameters.

This function sets the control parameters of GINT peripheral.

Parameters

- base – Base address of the GINT peripheral.
- comb – Controls if the enabled inputs are logically ORed or ANDed for interrupt generation.
- trig – Controls if the enabled inputs are level or edge sensitive based on polarity.
- callback – This function is called when configured group interrupt is generated.

Return values

None. –

void GINT_GetCtrl(GINT_Type *base, *gint_comb_t* *comb, *gint_trig_t* *trig, *gint_cb_t* *callback)

Get GINT peripheral control parameters.

This function returns the control parameters of GINT peripheral.

Parameters

- base – Base address of the GINT peripheral.
- comb – Pointer to store combine input value.
- trig – Pointer to store trigger value.
- callback – Pointer to store callback function.

Return values

None. –

void GINT_ConfigPins(GINT_Type *base, *gint_port_t* port, uint32_t polarityMask, uint32_t enableMask)

Configure GINT peripheral pins.

This function enables and controls the polarity of enabled pin(s) of a given port.

Parameters

- `base` – Base address of the GINT peripheral.
- `port` – Port number.
- `polarityMask` – Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- `enableMask` – Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT__GetConfigPins(GINT_Type *base, gint_port_t port, uint32_t *polarityMask, uint32_t *enableMask)
```

Get GINT peripheral pin configuration.

This function returns the pin configuration of a given port.

Parameters

- `base` – Base address of the GINT peripheral.
- `port` – Port number.
- `polarityMask` – Pointer to store the polarity mask Each bit position indicates the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- `enableMask` – Pointer to store the enable mask. Each bit position indicates if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT__EnableCallback(GINT_Type *base)
```

Enable callback.

This function enables the interrupt for the selected GINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

- `base` – Base address of the GINT peripheral.

Return values

None. –

```
void GINT__DisableCallback(GINT_Type *base)
```

Disable callback.

This function disables the interrupt for the selected GINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

- `base` – Base address of the peripheral.

Return values

None. –

```
static inline void GINT__ClrStatus(GINT_Type *base)
```

Clear GINT status.

This function clears the GINT status bit.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

```
static inline uint32_t GINT__GetStatus(GINT_Type *base)
```

Get GINT status.

This function returns the GINT status.

Parameters

- base – Base address of the GINT peripheral.

Return values

status – = 0 No group interrupt request. = 1 Group interrupt request active.

```
void GINT__Deinit(GINT_Type *base)
```

Deinitialize GINT peripheral.

This function disables the GINT clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

2.10 I2C: Inter-Integrated Circuit Driver

2.11 I2C DMA Driver

```
void I2C__MasterTransferCreateHandleDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                                         i2c_master_dma_transfer_callback_t callback, void  
                                         *userData, dma_handle_t *dmaHandle)
```

Init the I2C handle which is used in transactional functions.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- callback – pointer to user callback function
- userData – user param passed to the callback function
- dmaHandle – DMA handle pointer

```
status_t I2C__MasterTransferDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                                i2c_master_transfer_t *xfer)
```

Performs a master dma non-blocking transfer on the I2C bus.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- xfer – pointer to transfer structure of i2c_master_transfer_t

Return values

- `kStatus_Success` – Successfully complete the data transmission.
- `kStatus_I2C_Busy` – Previous transmission still not finished.
- `kStatus_I2C_Timeout` – Transfer error, wait signal timeout.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive Nak during transfer.

`status_t I2C_MasterTransferGetCountDMA(I2C_Type *base, i2c_master_dma_handle_t *handle, size_t *count)`

Get master transfer status during a dma non-blocking transfer.

Parameters

- `base` – I2C peripheral base address
- `handle` – pointer to `i2c_master_dma_handle_t` structure
- `count` – Number of bytes transferred so far by the non-blocking transaction.

`void I2C_MasterTransferAbortDMA(I2C_Type *base, i2c_master_dma_handle_t *handle)`

Abort a master dma non-blocking transfer in a early time.

Parameters

- `base` – I2C peripheral base address
- `handle` – pointer to `i2c_master_dma_handle_t` structure

`FSL_I2C_DMA_DRIVER_VERSION`

I2C DMA driver version.

`typedef struct i2c_master_dma_handle i2c_master_dma_handle_t`

I2C master dma handle typedef.

`typedef void (*i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)`

I2C master dma transfer callback typedef.

`typedef void (*flexcomm_i2c_dma_master_irq_handler_t)(I2C_Type *base, i2c_master_dma_handle_t *handle)`

Typedef for master dma handler.

`I2C_MAX_DMA_TRANSFER_COUNT`

Maximum length of single DMA transfer (determined by capability of the DMA engine)

`struct i2c_master_dma_handle`

`#include <fsl_i2c_dma.h>` I2C master dma transfer structure.

Public Members

`uint8_t state`

Transfer state machine current state.

`uint32_t transferCount`

Indicates progress of the transfer

`uint32_t remainingBytesDMA`

Remaining byte count to be transferred using DMA.

`uint8_t *buf`

Buffer pointer for current state.

`bool checkAddrNack`

Whether to check the nack signal is detected during addressing.

`dma_handle_t *dmaHandle`

The DMA handler used.

`i2c_master_transfer_t transfer`

Copy of the current transfer info.

`i2c_master_dma_transfer_callback_t completionCallback`

Callback function called after dma transfer finished.

`void *userData`

Callback parameter passed to callback function.

2.12 I2C Driver

`FSL_I2C_DRIVER_VERSION`

I2C driver version.

I2C status return codes.

Values:

`enumerator kStatus_I2C_Busy`

The master is already performing a transfer.

`enumerator kStatus_I2C_Idle`

The slave driver is idle.

`enumerator kStatus_I2C_Nak`

The slave device sent a NAK in response to a byte.

`enumerator kStatus_I2C_InvalidParameter`

Unable to proceed due to invalid parameter.

`enumerator kStatus_I2C_BitError`

Transferred bit was not seen on the bus.

`enumerator kStatus_I2C_ArbitrationLost`

Arbitration lost error.

`enumerator kStatus_I2C_NoTransferInProgress`

Attempt to abort a transfer when one is not in progress.

`enumerator kStatus_I2C_DmaRequestFail`

DMA request failed.

`enumerator kStatus_I2C_StartStopError`

Start and stop error.

`enumerator kStatus_I2C_UnexpectedState`

Unexpected state.

`enumerator kStatus_I2C_Timeout`

Timeout when waiting for I2C master/slave pending status to set to continue transfer.

`enumerator kStatus_I2C_Addr_Nak`

NAK received for Address

enumerator kStatus_I2C_EventTimeout

Timeout waiting for bus event.

enumerator kStatus_I2C_SclLowTimeout

Timeout SCL signal remains low.

enum __i2c_status_flags

I2C status flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingFlag

The I2C module is waiting for software interaction. bit 0

enumerator kI2C_MasterArbitrationLostFlag

The arbitration of the bus was lost. There was collision on the bus. bit 4

enumerator kI2C_MasterStartStopErrorFlag

There was an error during start or stop phase of the transaction. bit 6

enumerator kI2C_MasterIdleFlag

The I2C master idle status. bit 5

enumerator kI2C_MasterRxReadyFlag

The I2C master rx ready status. bit 1

enumerator kI2C_MasterTxReadyFlag

The I2C master tx ready status. bit 2

enumerator kI2C_MasterAddrNackFlag

The I2C master address nack status. bit 7

enumerator kI2C_MasterDataNackFlag

The I2C master data nack status. bit 3

enumerator kI2C_SlavePendingFlag

The I2C module is waiting for software interaction. bit 8

enumerator kI2C_SlaveNotStretching

Indicates whether the slave is currently stretching clock (0 = yes, 1 = no). bit 11

enumerator kI2C_SlaveSelected

Indicates whether the slave is selected by an address match. bit 14

enumerator kI2C_SlaveDeselected

Indicates that slave was previously deselected (deselect event took place, w1c). bit 15

enumerator kI2C_SlaveAddressedFlag

One of the I2C slave's 4 addresses is matched. bit 22

enumerator kI2C_SlaveReceiveFlag

Slave receive data available. bit 9

enumerator kI2C_SlaveTransmitFlag

Slave data can be transmitted. bit 10

enumerator kI2C_SlaveAddress0MatchFlag

Slave address0 match. bit 20

enumerator kI2C_SlaveAddress1MatchFlag
Slave address1 match. bit 12

enumerator kI2C_SlaveAddress2MatchFlag
Slave address2 match. bit 13

enumerator kI2C_SlaveAddress3MatchFlag
Slave address3 match. bit 21

enumerator kI2C_MonitorReadyFlag
The I2C monitor ready interrupt. bit 16

enumerator kI2C_MonitorOverflowFlag
The monitor data overrun interrupt. bit 17

enumerator kI2C_MonitorActiveFlag
The monitor is active. bit 18

enumerator kI2C_MonitorIdleFlag
The monitor idle interrupt. bit 19

enumerator kI2C_EventTimeoutFlag
The bus event timeout interrupt. bit 24

enumerator kI2C_SclTimeoutFlag
The SCL timeout interrupt. bit 25

enumerator kI2C_MasterAllClearFlags

enumerator kI2C_SlaveAllClearFlags

enumerator kI2C_CommonAllClearFlags

enum _i2c_interrupt_enable
I2C interrupt enable.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingInterruptEnable
The I2C master communication pending interrupt.

enumerator kI2C_MasterArbitrationLostInterruptEnable
The I2C master arbitration lost interrupt.

enumerator kI2C_MasterStartStopErrorInterruptEnable
The I2C master start/stop timing error interrupt.

enumerator kI2C_SlavePendingInterruptEnable
The I2C slave communication pending interrupt.

enumerator kI2C_SlaveNotStretchingInterruptEnable
The I2C slave not stretching interrupt, deep-sleep mode can be entered only when this interrupt occurs.

enumerator kI2C_SlaveDeselectedInterruptEnable
The I2C slave deselection interrupt.

enumerator kI2C_MonitorReadyInterruptEnable
The I2C monitor ready interrupt.

enumerator `kI2C_MonitorOverflowInterruptEnable`

The monitor data overrun interrupt.

enumerator `kI2C_MonitorIdleInterruptEnable`

The monitor idle interrupt.

enumerator `kI2C_EventTimeoutInterruptEnable`

The bus event timeout interrupt.

enumerator `kI2C_SclTimeoutInterruptEnable`

The SCL timeout interrupt.

enumerator `kI2C_MasterAllInterruptEnable`

enumerator `kI2C_SlaveAllInterruptEnable`

enumerator `kI2C_CommonAllInterruptEnable`

`I2C_RETRY_TIMES`

Retry times for waiting flag.

`I2C_MASTER_TRANSMIT_IGNORE_LAST_NACK`

Whether to ignore the nack signal of the last byte during master transmit.

`I2C_STAT_MSTCODE_IDLE`

Master Idle State Code

`I2C_STAT_MSTCODE_RXREADY`

Master Receive Ready State Code

`I2C_STAT_MSTCODE_TXREADY`

Master Transmit Ready State Code

`I2C_STAT_MSTCODE_NACKADR`

Master NACK by slave on address State Code

`I2C_STAT_MSTCODE_NACKDAT`

Master NACK by slave on data State Code

`I2C_STAT_SLVST_ADDR`

`I2C_STAT_SLVST_RX`

`I2C_STAT_SLVST_TX`

2.13 I2C Master Driver

`void I2C_MasterGetDefaultConfig(i2c_master_config_t *masterConfig)`

Provides a default configuration for the I2C master peripheral.

This function provides the following default configuration for the I2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->baudRate_Bps      = 100000U;
masterConfig->enableTimeout     = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i2c_master_config_t`.

```
void I2C_MasterInit(I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
```

Initializes the I2C master peripheral.

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `srcClock_Hz` – Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

```
void I2C_MasterDeinit(I2C_Type *base)
```

Deinitializes the I2C master peripheral.

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

```
uint32_t I2C_GetInstance(I2C_Type *base)
```

Returns an instance number given a base address.

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

- `base` – The I2C peripheral base address.

Returns

I2C instance number starting from 0.

```
static inline void I2C_MasterReset(I2C_Type *base)
```

Performs a software reset.

Restores the I2C master peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

```
static inline void I2C_MasterEnable(I2C_Type *base, bool enable)
```

Enables or disables the I2C module as master.

Parameters

- `base` – The I2C peripheral base address.
- `enable` – Pass true to enable or false to disable the specified I2C as master.

```
uint32_t I2C_GetStatusFlags(I2C_Type *base)
```

Gets the I2C status flags.

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i2c_status_flags`.

Parameters

- `base` – The I2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I2C_ClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C status flag state.

Refer to `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags` to see the clearable flags. Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`, `_i2c_master_status_flags` and `_i2c_slave_status_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of the members in `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags`. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

```
static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C master status flag state.

Deprecated:

Do not use this function. It has been superseded by `I2C_ClearStatusFlags`. The following status register flags can be cleared:

- `kI2C_MasterArbitrationLostFlag`
- `kI2C_MasterStartStopErrorFlag`

Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_status_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

```
static inline void I2C__EnableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Enables the I2C interrupt requests.

Parameters

- base – The I2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I2C__DisableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Disables the I2C interrupt requests.

Parameters

- base – The I2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I2C__GetEnabledInterrupts(I2C_Type *base)
```

Returns the set of currently enabled I2C interrupt requests.

Parameters

- base – The I2C peripheral base address.

Returns

A bitmask composed of `_i2c_interrupt_enable` enumerators OR'd together to indicate the set of enabled interrupts.

```
void I2C__MasterSetBaudRate(I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
```

Sets the I2C bus frequency for master transactions.

The I2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

- base – The I2C peripheral base address.
- srcClock_Hz – I2C functional clock frequency in Hertz.
- baudRate_Bps – Requested bus frequency in bits per second.

```
void I2C__MasterSetTimeoutValue(I2C_Type *base, uint8_t timeout_Ms, uint32_t srcClock_Hz)
```

Sets the I2C bus timeout value.

If the SCL signal remains low or bus does not have event longer than the timeout value, `kI2C_SclTimeoutFlag` or `kI2C_EventTimeoutFlag` is set. This can indicate the bus is held by slave or any fault occurs to the I2C module.

Parameters

- base – The I2C peripheral base address.
- timeout_Ms – Timeout value in millisecond.
- srcClock_Hz – I2C functional clock frequency in Hertz.

```
static inline bool I2C__MasterGetBusIdleState(I2C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- base – The I2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

status_t I2C_MasterStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)

Sends a START on the I2C bus.

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

- base – I2C peripheral base pointer
- address – 7-bit slave device address.
- direction – Master transfer directions(transmit/receive).

Return values

- kStatus_Success – Successfully send the start signal.
- kStatus_I2C_Busy – Current bus is busy.

status_t I2C_MasterStop(I2C_Type *base)

Sends a STOP signal on the I2C bus.

Return values

- kStatus_Success – Successfully send the stop signal.
- kStatus_I2C_Timeout – Send stop signal failed, timeout.

static inline *status_t* I2C_MasterRepeatedStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)

Sends a REPEATED START on the I2C bus.

Parameters

- base – I2C peripheral base pointer
- address – 7-bit slave device address.
- direction – Master transfer directions(transmit/receive).

Return values

- kStatus_Success – Successfully send the start signal.
- kStatus_I2C_Busy – Current bus is busy but not occupied by current I2C master.

status_t I2C_MasterWriteBlocking(I2C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_I2C_Nak.

Parameters

- base – The I2C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- flags – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_I2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_I2C_ArbitrationLost` – Arbitration lost error.

`status_t I2C_MasterReadBlocking(I2C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)`

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The I2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.
- `flags` – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use `kI2C_TransferDefaultFlag`

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_I2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_I2C_ArbitrationLost` – Arbitration lost error.

`status_t I2C_MasterTransferBlocking(I2C_Type *base, i2c_master_transfer_t *xfer)`

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

- `base` – I2C peripheral base address.
- `xfer` – Pointer to the transfer structure.

Return values

- `kStatus_Success` – Successfully complete the data transmission.
- `kStatus_I2C_Busy` – Previous transmission still not finished.
- `kStatus_I2C_Timeout` – Transfer error, wait signal timeout.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.
- `kStatus_I2C_Addr_Nak` – Transfer error, receive NAK during addressing.

`void I2C_MasterTransferCreateHandle(I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_callback_t callback, void *userData)`

Creates a new handle for the I2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I2C_MasterTransferAbort()` API shall be called.

Parameters

- `base` – The I2C peripheral base address.

- `handle` – **[out]** Pointer to the I2C master driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I2C_MasterTransferNonBlocking(I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_t *xfer)`

Performs a non-blocking transaction on the I2C bus.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.
- `xfer` – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I2C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

`status_t I2C_MasterTransferGetCount(I2C_Type *base, i2c_master_handle_t *handle, size_t *count)`

Returns number of bytes transferred so far.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.
- `count` – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_Success` –
- `kStatus_I2C_Busy` –

`status_t I2C_MasterTransferAbort(I2C_Type *base, i2c_master_handle_t *handle)`

Terminates a non-blocking I2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.

Return values

- `kStatus_Success` – A transaction was successfully aborted.
- `kStatus_I2C_Timeout` – Timeout during polling for flags.

`void I2C_MasterTransferHandleIRQ(I2C_Type *base, i2c_master_handle_t *handle)`

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I2C peripheral base address.
- handle – Pointer to the I2C master driver handle.

enum `_i2c_direction`

Direction of master and slave transfers.

Values:

enumerator `kI2C_Write`

Master transmit.

enumerator `kI2C_Read`

Master receive.

enum `_i2c_master_transfer_flags`

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_i2c_master_transfer::flags` field.

Values:

enumerator `kI2C_TransferDefaultFlag`

Transfer starts with a start signal, stops with a stop signal.

enumerator `kI2C_TransferNoStartFlag`

Don't send a start condition, address, and sub address

enumerator `kI2C_TransferRepeatedStartFlag`

Send a repeated start condition

enumerator `kI2C_TransferNoStopFlag`

Don't send a stop condition.

enum `_i2c_transfer_states`

States for the state machine used by transactional APIs.

Values:

enumerator `kIdleState`

enumerator `kTransmitSubaddrState`

enumerator `kTransmitDataState`

enumerator `kReceiveDataBeginState`

enumerator `kReceiveDataState`

enumerator `kReceiveLastDataState`

enumerator `kStartState`

enumerator `kStopState`

enumerator `kWaitForCompletionState`

typedef enum `_i2c_direction` `i2c_direction_t`

Direction of master and slave transfers.

```
typedef struct _i2c_master_config i2c_master_config_t
```

Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef struct _i2c_master_transfer i2c_master_transfer_t
```

I2C master transfer typedef.

```
typedef struct _i2c_master_handle i2c_master_handle_t
```

I2C master handle typedef.

```
typedef void (*i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle,
status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to I2C_MasterTransferCreateHandle().

Param base

The I2C peripheral base address.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
struct _i2c_master_config
```

#include <fsl_i2c.h> Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

uint32_t baudRate_Bps

Desired baud rate in bits per second.

bool enableTimeout

Enable internal timeout function.

uint8_t timeout_Ms

Event timeout and SCL low timeout value.

```
struct _i2c_master_transfer
```

#include <fsl_i2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_i2c_master_transfer_flags` for available options. Set to 0 or `ki2C_TransferDefaultFlag` for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i2c_direction_t direction

Either `ki2C_Read` or `ki2C_Write`.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct `_i2c_master_handle`

#include <fsl_i2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint32_t transferCount

Indicates progress of the transfer

uint32_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

bool checkAddrNack

Whether to check the nack signal is detected during addressing.

i2c_master_transfer_t transfer

Copy of the current transfer info.

i2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData

Application data passed to callback.

2.14 I2C Slave Driver

`void I2C_SlaveGetDefaultConfig(i2c_slave_config_t *slaveConfig)`

Provides a default configuration for the I2C slave peripheral.

This function provides the following default configuration for the I2C slave peripheral:

```
slaveConfig->enableSlave = true;
slaveConfig->address0.disable = false;
slaveConfig->address0.address = 0u;
slaveConfig->address1.disable = true;
slaveConfig->address2.disable = true;
slaveConfig->address3.disable = true;
slaveConfig->busSpeed = kI2C_SlaveStandardMode;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `I2C_SlaveInit()`. Be sure to override at least the *address0.address* member of the configuration structure with the desired slave address.

Parameters

- *slaveConfig* – **[out]** User provided configuration structure that is set to default values. Refer to *i2c_slave_config_t*.

`status_t I2C_SlaveInit(I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t srcClock_Hz)`

Initializes the I2C slave peripheral.

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

- *base* – The I2C peripheral base address.
- *slaveConfig* – User provided peripheral configuration. Use `I2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- *srcClock_Hz* – Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

`void I2C_SlaveSetAddress(I2C_Type *base, i2c_slave_address_register_t addressRegister, uint8_t address, bool addressDisable)`

Configures Slave Address n register.

This function writes new value to Slave Address register.

Parameters

- *base* – The I2C peripheral base address.
- *addressRegister* – The module supports multiple address registers. The parameter determines which one shall be changed.
- *address* – The slave address to be stored to the address register for matching.
- *addressDisable* – Disable matching of the specified address register.

`void I2C_SlaveDeinit(I2C_Type *base)`

Deinitializes the I2C slave peripheral.

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

`static inline void I2C__SlaveEnable(I2C_Type *base, bool enable)`

Enables or disables the I2C module as slave.

Parameters

- `base` – The I2C peripheral base address.
- `enable` – True to enable or false to disable.

`static inline void I2C__SlaveClearStatusFlags(I2C_Type *base, uint32_t statusMask)`

Clears the I2C status flag state.

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

See also:

`_i2c_slave_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_SlaveGetStatusFlags()`.

`status_t I2C__SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)`

Performs a polling send transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- `base` – The I2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

`kStatus_Success` Data has been sent.

Returns

`kStatus_Fail` Unexpected slave state (master data write while master read from slave is expected).

`status_t I2C__SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- `base` – The I2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

`kStatus_Success` Data has been received.

Returns

kStatus_Fail Unexpected slave state (master data read while master write to slave is expected).

```
void I2C_SlaveTransferCreateHandle(I2C_Type *base, i2c_slave_handle_t *handle,  
                                i2c_slave_transfer_callback_t callback, void *userData)
```

Creates a new handle for the I2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I2C_SlaveTransferAbort() API shall be called.

Parameters

- base – The I2C peripheral base address.
- handle – **[out]** Pointer to the I2C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

```
status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, i2c_slave_handle_t *handle, uint32_t  
                                     eventMask)
```

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes kI2C_SlaveTransmitEvent callback. If no slave Rx transfer is busy, a master write to slave request invokes kI2C_SlaveReceiveEvent callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of *i2c_slave_transfer_event_t* enumerators for the events you wish to receive. The *kI2C_SlaveTransmitEvent* and *kI2C_SlaveReceiveEvent* events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the *kI2C_SlaveAllEvents* constant is provided as a convenient way to enable all events.

Parameters

- base – The I2C peripheral base address.
- handle – Pointer to *i2c_slave_handle_t* structure which stores the transfer state.
- eventMask – Bit mask formed by OR'ing together *i2c_slave_transfer_event_t* enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and *kI2C_SlaveAllEvents* to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.
- kStatus_I2C_Busy – Slave transfers have already been started on this handle.

```
status_t I2C_SlaveSetSendBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, const  
                               void *txData, size_t txSize, uint32_t eventMask)
```

Starts accepting master read from slave requests.

The function can be called in response to `ki2c_SlaveTransmitEvent` callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `ki2c_SlaveTransmitEvent` and `ki2c_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `ki2c_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `txData` – Pointer to data to send to master.
- `txSize` – Size of `txData` in bytes.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `ki2c_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`status_t I2C_SlaveSetReceiveBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *rxData, size_t rxSize, uint32_t eventMask)`

Starts accepting master write to slave requests.

The function can be called in response to `ki2c_SlaveReceiveEvent` callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `ki2c_SlaveTransmitEvent` and `ki2c_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `ki2c_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `rxData` – Pointer to data to store data from master.
- `rxSize` – Size of `rxData` in bytes.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `ki2c_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

```
static inline uint32_t I2C_SlaveGetReceivedAddress(I2C_Type *base, volatile i2c_slave_transfer_t *transfer)
```

Returns the slave address sent by the I2C master.

This function should only be called from the address match event callback `kI2C_SlaveAddressMatchEvent`.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

```
void I2C_SlaveTransferAbort(I2C_Type *base, i2c_slave_handle_t *handle)
```

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

Return values

- `kStatus_Success` –
- `kStatus_I2C_Idle` –

```
status_t I2C_SlaveTransferGetCount(I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
```

Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

```
void I2C_SlaveTransferHandleIRQ(I2C_Type *base, i2c_slave_handle_t *handle)
```

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

enum `_i2c_slave_address_register`

I2C slave address register.

Values:

enumerator `kI2C_SlaveAddressRegister0`

Slave Address 0 register.

enumerator `kI2C_SlaveAddressRegister1`

Slave Address 1 register.

enumerator `kI2C_SlaveAddressRegister2`

Slave Address 2 register.

enumerator `kI2C_SlaveAddressRegister3`

Slave Address 3 register.

enum `_i2c_slave_address_qual_mode`

I2C slave address match options.

Values:

enumerator `kI2C_QualModeMask`

The SLVQUAL0 field (`qualAddress`) is used as a logical mask for matching address0.

enumerator `kI2C_QualModeExtend`

The SLVQUAL0 (`qualAddress`) field is used to extend address 0 matching in a range of addresses.

enum `_i2c_slave_bus_speed`

I2C slave bus speed options.

Values:

enumerator `kI2C_SlaveStandardMode`

enumerator `kI2C_SlaveFastMode`

enumerator `kI2C_SlaveFastModePlus`

enumerator `kI2C_SlaveHsMode`

enum `_i2c_slave_transfer_event`

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator `kI2C_SlaveAddressMatchEvent`

Received the slave address after a start or repeated start.

enumerator `kI2C_SlaveTransmitEvent`

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator `kI2C_SlaveReceiveEvent`

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator `kI2C_SlaveCompletionEvent`

All data in the active transfer have been consumed.

enumerator `kI2C_SlaveDeselectedEvent`

The slave function has become deselected (SLVSEL flag changing from 1 to 0).

enumerator `kI2C_SlaveAllEvents`

Bit mask of all available events.

enum `_i2c_slave_fsm`

I2C slave software finite state machine states.

Values:

enumerator `kI2C_SlaveFsmAddressMatch`

enumerator `kI2C_SlaveFsmReceive`

enumerator `kI2C_SlaveFsmTransmit`

typedef enum `_i2c_slave_address_register` `i2c_slave_address_register_t`

I2C slave address register.

typedef struct `_i2c_slave_address` `i2c_slave_address_t`

Data structure with 7-bit Slave address and Slave address disable.

typedef enum `_i2c_slave_address_qual_mode` `i2c_slave_address_qual_mode_t`

I2C slave address match options.

typedef enum `_i2c_slave_bus_speed` `i2c_slave_bus_speed_t`

I2C slave bus speed options.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`

Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the `I2C_SlaveGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum `_i2c_slave_transfer_event` `i2c_slave_transfer_event_t`

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct `_i2c_slave_handle` `i2c_slave_handle_t`

I2C slave handle typedef.

typedef struct `_i2c_slave_transfer` `i2c_slave_transfer_t`

I2C slave transfer structure.

typedef void (*`i2c_slave_transfer_callback_t`)(`I2C_Type` *base, volatile `i2c_slave_transfer_t` *transfer, void *userData)

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the `I2C_SlaveSetCallback()` function after you have created a handle.

Param base

Base address for the I2C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the call-back.

Param userData

Arbitrary pointer-sized value passed from the application.

```
typedef enum _i2c_slave_fsm i2c_slave_fsm_t
```

I2C slave software finite state machine states.

```
typedef void (*flexcomm_i2c_master_irq_handler_t)(I2C_Type *base, i2c_master_handle_t *handle)
```

Typedef for master interrupt handler.

```
typedef void (*flexcomm_i2c_slave_irq_handler_t)(I2C_Type *base, i2c_slave_handle_t *handle)
```

Typedef for slave interrupt handler.

```
struct _i2c_slave_address
```

#include <fsl_i2c.h> Data structure with 7-bit Slave address and Slave address disable.

Public Members

uint8_t address

7-bit Slave address SLVADR.

bool addressDisable

Slave address disable SADISABLE.

```
struct _i2c_slave_config
```

#include <fsl_i2c.h> Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i2c_slave_address_t address0

Slave's 7-bit address and disable.

i2c_slave_address_t address1

Alternate slave 7-bit address and disable.

i2c_slave_address_t address2

Alternate slave 7-bit address and disable.

i2c_slave_address_t address3

Alternate slave 7-bit address and disable.

i2c_slave_address_qual_mode_t qualMode

Qualify mode for slave address 0.

uint8_t qualAddress

Slave address qualifier for address 0.

i2c_slave_bus_speed_t busSpeed

Slave bus speed mode. If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The busSpeed value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the busSpeed mode is unknown at compile time, use the longest data setup time kI2C_SlaveStandardMode (250 ns)

bool enableSlave

Enable slave mode.

struct *_i2c_slave_transfer*

#include <fsl_i2c.h> I2C slave transfer structure.

Public Members

i2c_slave_handle_t *handle

Pointer to handle that contains this transfer.

i2c_slave_transfer_event_t event

Reason the callback is being invoked.

uint8_t receivedAddress

Matching address send by master. 7-bits plus R/nW bit0

uint32_t eventMask

Mask of enabled events.

uint8_t *rxData

Transfer buffer for receive data

const uint8_t *txData

Transfer buffer for transmit data

size_t txSize

Transfer size

size_t rxSize

Transfer size

size_t transferredCount

Number of bytes transferred during this transfer.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI2C_SlaveCompletionEvent.

struct *_i2c_slave_handle*

#include <fsl_i2c.h> I2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

volatile *i2c_slave_transfer_t* transfer

I2C slave transfer.

volatile bool isBusy

Whether transfer is busy.

volatile *i2c_slave_fsm_t* slaveFsm

slave transfer state machine.

i2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

2.15 I2S: I2S Driver

2.16 I2S DMA Driver

```
void I2S_TxTransferCreateHandleDMA(I2S_Type *base, i2s_dma_handle_t *handle, dma_handle_t
                                *dmaHandle, i2s_dma_transfer_callback_t callback, void
                                *userData)
```

Initializes handle for transfer of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- dmaHandle – pointer to dma handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

```
status_t I2S_TxTransferSendDMA(I2S_Type *base, i2s_dma_handle_t *handle, i2s_transfer_t
                                transfer)
```

Begins or queue sending of the given data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus_Success –
- kStatus_I2S_Busy – if all queue slots are occupied with unsent buffers.

```
void I2S_TransferAbortDMA(I2S_Type *base, i2s_dma_handle_t *handle)
```

Aborts transfer of data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

```
void I2S_RxTransferCreateHandleDMA(I2S_Type *base, i2s_dma_handle_t *handle, dma_handle_t
                                *dmaHandle, i2s_dma_transfer_callback_t callback, void
                                *userData)
```

Initializes handle for reception of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- dmaHandle – pointer to dma handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_RxTransferReceiveDMA(I2S_Type *base, *i2s_dma_handle_t* *handle, *i2s_transfer_t* transfer)

Begins or queue reception of data into given buffer.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus_Success –
- kStatus_I2S_Busy – if all queue slots are occupied with buffers which are not full.

void I2S_DMACallback(*dma_handle_t* *handle, *void* *userData, *bool* transferDone, *uint32_t* tcds)

Invoked from DMA interrupt handler.

Parameters

- handle – pointer to DMA handle structure.
- userData – argument for user callback.
- transferDone – if transfer was done.
- tcds –

void I2S_TransferInstallLoopDMADescriptorMemory(*i2s_dma_handle_t* *handle, *void* *dmaDescriptorAddr, *size_t* dmaDescriptorNum)

Install DMA descriptor memory for loop transfer only.

This function used to register DMA descriptor memory for the i2s loop dma transfer.

It must be called before I2S_TransferSendLoopDMA/I2S_TransferReceiveLoopDMA and after I2S_RxTransferCreateHandleDMA/I2S_TxTransferCreateHandleDMA.

User should be take care about the address of DMA descriptor pool which required align with 16BYTE at least.

Parameters

- handle – Pointer to i2s DMA transfer handle.
- dmaDescriptorAddr – DMA descriptor start address.
- dmaDescriptorNum – DMA descriptor number.

status_t I2S_TransferSendLoopDMA(I2S_Type *base, *i2s_dma_handle_t* *handle, *i2s_transfer_t* *xfer, *uint32_t* loopTransferCount)

Send link transfer data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

This function support loop transfer, such as A->B->...->A, the loop transfer chain will be converted into a chain of descriptor and submit to dma. Application must be aware of that the more counts of the loop transfer, then more DMA descriptor memory required, user can use function `I2S_InstallDMADescriptorMemory` to register the dma descriptor memory.

As the DMA support maximum 1024 transfer count, so application must be aware of that this transfer function support maximum 1024 samples in each transfer, otherwise assert error or error status will be returned. Once the loop transfer start, application can use function `I2S_TransferAbortDMA` to stop the loop transfer.

Parameters

- base – I2S peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – I2S DMA transfer structure. See `i2s_transfer_t`.
- loopTransferCount – loop count

Return values

`kStatus_Success` –

`status_t I2S_TransferReceiveLoopDMA(I2S_Type *base, i2s_dma_handle_t *handle, i2s_transfer_t *xfer, uint32_t loopTransferCount)`

Receive link transfer data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

This function support loop transfer, such as A->B->...->A, the loop transfer chain will be converted into a chain of descriptor and submit to dma. Application must be aware of that the more counts of the loop transfer, then more DMA descriptor memory required, user can use function `I2S_InstallDMADescriptorMemory` to register the dma descriptor memory.

As the DMA support maximum 1024 transfer count, so application must be aware of that this transfer function support maximum 1024 samples in each transfer, otherwise assert error or error status will be returned. Once the loop transfer start, application can use function `I2S_TransferAbortDMA` to stop the loop transfer.

Parameters

- base – I2S peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – I2S DMA transfer structure. See `i2s_transfer_t`.
- loopTransferCount – loop count

Return values

`kStatus_Success` –

`FSL_I2S_DMA_DRIVER_VERSION`

I2S DMA driver version 2.3.3.

`typedef struct i2s_dma_handle i2s_dma_handle_t`

Members not to be accessed / modified outside of the driver.

`typedef void (*i2s_dma_transfer_callback_t)(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)`

Callback function invoked from DMA API on completion.

Param base

I2S base pointer.

Param handle

pointer to I2S transaction.

Param completionStatus
status of the transaction.

Param userData
optional pointer to user arguments data.

```
struct __i2s_dma_handle  
#include <fsl_i2s_dma.h> i2s_dma_handle
```

Public Members

```
uint32_t state  
    Internal state of I2S DMA transfer  
uint8_t bytesPerFrame  
    bytes per frame  
i2s_dma_transfer_callback_t completionCallback  
    Callback function pointer  
void *userData  
    Application data passed to callback  
dma_handle_t *dmaHandle  
    DMA handle  
volatile i2s_transfer_t i2sQueue[(4U)]  
    Transfer queue storing transfer buffers  
volatile uint8_t queueUser  
    Queue index where user's next transfer will be stored  
volatile uint8_t queueDriver  
    Queue index of buffer actually used by the driver  
dma_descriptor_t *i2sLoopDMADescriptor  
    descriptor pool pointer  
size_t i2sLoopDMADescriptorNum  
    number of descriptor in descriptors pool
```

2.17 I2S Driver

```
void I2S_TxInit(I2S_Type *base, const i2s_config_t *config)
```

Initializes the FLEXCOMM peripheral for I2S transmit functionality.

Ungates the FLEXCOMM clock and configures the module for I2S transmission using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_TxGetDefaultConfig().

Note: This API should be called at the beginning of the application to use the I2S driver.

Parameters

- base – I2S base pointer.
- config – pointer to I2S configuration structure.

```
void I2S_RxInit(I2S_Type *base, const i2s_config_t *config)
```

Initializes the FLEXCOMM peripheral for I2S receive functionality.

Ungates the FLEXCOMM clock and configures the module for I2S receive using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_RxGetDefaultConfig().

Note: This API should be called at the beginning of the application to use the I2S driver.

Parameters

- base – I2S base pointer.
- config – pointer to I2S configuration structure.

```
void I2S_TxGetDefaultConfig(i2s_config_t *config)
```

Sets the I2S Tx configuration structure to default values.

This API initializes the configuration structure for use in I2S_TxInit(). The initialized structure can remain unchanged in I2S_TxInit(), or it can be modified before calling I2S_TxInit(). Example:

```
i2s_config_t config;  
I2S_TxGetDefaultConfig(&config);
```

Default values:

```
config->masterSlave = kI2S_MasterSlaveNormalMaster;  
config->mode = kI2S_ModeI2sClassic;  
config->rightLow = false;  
config->leftJust = false;  
config->pdmData = false;  
config->sckPol = false;  
config->wsPol = false;  
config->divider = 1;  
config->oneChannel = false;  
config->dataLength = 16;  
config->frameLength = 32;  
config->position = 0;  
config->watermark = 4;  
config->txEmptyZero = true;  
config->pack48 = false;
```

Parameters

- config – pointer to I2S configuration structure.

```
void I2S_RxGetDefaultConfig(i2s_config_t *config)
```

Sets the I2S Rx configuration structure to default values.

This API initializes the configuration structure for use in I2S_RxInit(). The initialized structure can remain unchanged in I2S_RxInit(), or it can be modified before calling I2S_RxInit(). Example:

```
i2s_config_t config;  
I2S_RxGetDefaultConfig(&config);
```

Default values:

```
config->masterSlave = kI2S_MasterSlaveNormalSlave;  
config->mode = kI2S_ModeI2sClassic;
```

(continues on next page)

(continued from previous page)

```

config->rightLow = false;
config->leftJust = false;
config->pdmData = false;
config->sckPol = false;
config->wsPol = false;
config->divider = 1;
config->oneChannel = false;
config->dataLength = 16;
config->frameLength = 32;
config->position = 0;
config->watermark = 4;
config->txEmptyZero = false;
config->pack48 = false;

```

Parameters

- config – pointer to I2S configuration structure.

void I2S_Deinit(I2S_Type *base)

De-initializes the I2S peripheral.

This API gates the FLEXCOMM clock. The I2S module can't operate unless I2S_TxInit or I2S_RxInit is called to enable the clock.

Parameters

- base – I2S base pointer.

void I2S_SetBitClockRate(I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)

Transmitter/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 I2S_TxTransferCreateHandle(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_callback_t callback, void *userData)

Initializes handle for transfer of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_TxTransferNonBlocking(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_t transfer)

Begins or queue sending of the given data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

- transfer – data buffer.

Return values

- kStatus__Success –
- kStatus_I2S_Busy – if all queue slots are occupied with unsent buffers.

void I2S_TxTransferAbort(I2S_Type *base, i2s_handle_t *handle)

Aborts sending of data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

void I2S_RxTransferCreateHandle(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_callback_t callback, void *userData)

Initializes handle for reception of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_RxTransferNonBlocking(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_t transfer)

Begins or queue reception of data into given buffer.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus__Success –
- kStatus_I2S_Busy – if all queue slots are occupied with buffers which are not full.

void I2S_RxTransferAbort(I2S_Type *base, i2s_handle_t *handle)

Aborts receiving of data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

status_t I2S_TransferGetCount(I2S_Type *base, i2s_handle_t *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- count – **[out]** number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus__Success –
- kStatus_NoTransferInProgress – there is no non-blocking transaction currently in progress.

status_t I2S_TransferGetErrorCount(I2S_Type *base, *i2s_handle_t* *handle, *size_t* *count)

Returns number of buffer underruns or overruns.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- count – **[out]** number of transmit errors encountered so far by the non-blocking transaction.

Return values

- kStatus__Success –
- kStatus_NoTransferInProgress – there is no non-blocking transaction currently in progress.

static inline void I2S_Enable(I2S_Type *base)

Enables I2S operation.

Parameters

- base – I2S base pointer.

void I2S_EnableSecondaryChannel(I2S_Type *base, *uint32_t* channel, *bool* oneChannel, *uint32_t* position)

Enables I2S secondary channel.

Parameters

- base – I2S base pointer.
- channel – secondary channel channel number, reference `_i2s_secondary_channel`.
- oneChannel – true is treated as single channel, functionality left channel for this pair.
- position – define the location within the frame of the data, should not bigger than 0x1FFU.

static inline void I2S_DisableSecondaryChannel(I2S_Type *base, *uint32_t* channel)

Disables I2S secondary channel.

Parameters

- base – I2S base pointer.
- channel – secondary channel channel number, reference `_i2s_secondary_channel`.

static inline void I2S_Disable(I2S_Type *base)

Disables I2S operation.

Parameters

- base – I2S base pointer.

static inline void I2S_EnableInterrupts(I2S_Type *base, *uint32_t* interruptMask)

Enables I2S FIFO interrupts.

Parameters

- base – I2S base pointer.

- `interruptMask` – bit mask of interrupts to enable. See `i2s_flags_t` for the set of constants that should be OR'd together to form the bit mask.

`static inline void I2S_DisableInterrupts(I2S_Type *base, uint32_t interruptMask)`

Disables I2S FIFO interrupts.

Parameters

- `base` – I2S base pointer.
- `interruptMask` – bit mask of interrupts to enable. See `i2s_flags_t` for the set of constants that should be OR'd together to form the bit mask.

`static inline uint32_t I2S_GetEnabledInterrupts(I2S_Type *base)`

Returns the set of currently enabled I2S FIFO interrupts.

Parameters

- `base` – I2S base pointer.

Returns

A bitmask composed of `i2s_flags_t` enumerators OR'd together to indicate the set of enabled interrupts.

`status_t I2S_EmptyTxFifo(I2S_Type *base)`

Flush the valid data in TX fifo.

Parameters

- `base` – I2S base pointer.

Returns

`kStatus_Fail` empty TX fifo failed, `kStatus_Success` empty tx fifo success.

`void I2S_TxHandleIRQ(I2S_Type *base, i2s_handle_t *handle)`

Invoked from interrupt handler when transmit FIFO level decreases.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.

`void I2S_RxHandleIRQ(I2S_Type *base, i2s_handle_t *handle)`

Invoked from interrupt handler when receive FIFO level decreases.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.

`FSL_I2S_DRIVER_VERSION`

I2S driver version 2.3.2.

`_i2s_status` I2S status codes.

Values:

enumerator `kStatus_I2S_BufferComplete`

Transfer from/into a single buffer has completed

enumerator `kStatus_I2S_Done`

All buffers transfers have completed

enumerator `kStatus_I2S_Busy`

Already performing a transfer and cannot queue another buffer

enum _i2s_flags
I2S flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2S_TxErrorFlag
TX error interrupt

enumerator kI2S_TxLevelFlag
TX level interrupt

enumerator kI2S_RxErrorFlag
RX error interrupt

enumerator kI2S_RxLevelFlag
RX level interrupt

enum _i2s_master_slave
Master / slave mode.

Values:

enumerator kI2S_MasterSlaveNormalSlave
Normal slave

enumerator kI2S_MasterSlaveWsSyncMaster
WS synchronized master

enumerator kI2S_MasterSlaveExtSckMaster
Master using existing SCK

enumerator kI2S_MasterSlaveNormalMaster
Normal master

enum _i2s_mode
I2S mode.

Values:

enumerator kI2S_ModeI2sClassic
I2S classic mode

enumerator kI2S_ModeDspWs50
DSP mode, WS having 50% duty cycle

enumerator kI2S_ModeDspWsShort
DSP mode, WS having one clock long pulse

enumerator kI2S_ModeDspWsLong
DSP mode, WS having one data slot long pulse

_i2s_secondary_channel I2S secondary channel.

Values:

enumerator kI2S_SecondaryChannel1
secondary channel 1

enumerator `kI2S_SecondaryChannel2`
secondary channel 2

enumerator `kI2S_SecondaryChannel3`
secondary channel 3

typedef enum `_i2s_flags` `i2s_flags_t`
I2S flags.

Note: These enums are meant to be OR'd together to form a bit mask.

typedef enum `_i2s_master_slave` `i2s_master_slave_t`
Master / slave mode.

typedef enum `_i2s_mode` `i2s_mode_t`
I2S mode.

typedef struct `_i2s_config` `i2s_config_t`
I2S configuration structure.

typedef struct `_i2s_transfer` `i2s_transfer_t`
Buffer to transfer from or receive audio data into.

typedef struct `_i2s_handle` `i2s_handle_t`
Transactional state of the initialized transfer or receive I2S operation.

typedef void (`*i2s_transfer_callback_t`)(`I2S_Type *base`, `i2s_handle_t *handle`, `status_t` `completionStatus`, void `*userData`)

Callback function invoked from transactional API on completion of a single buffer transfer.

Param base
I2S base pointer.

Param handle
pointer to I2S transaction.

Param completionStatus
status of the transaction.

Param userData
optional pointer to user arguments data.

`I2S_NUM_BUFFERS`
Number of buffers .

struct `_i2s_config`
`#include <fsl_i2s.h>` I2S configuration structure.

Public Members

`i2s_master_slave_t` `masterSlave`
Master / slave configuration

`i2s_mode_t` `mode`
I2S mode

bool `rightLow`
Right channel data in low portion of FIFO

bool leftJust

Left justify data in FIFO

bool pdmData

Data source is the D-Mic subsystem

bool sckPol

SCK polarity

bool wsPol

WS polarity

uint16_t divider

Flexcomm function clock divider (1 - 4096)

bool oneChannel

true mono, false stereo

uint8_t dataLength

Data length (4 - 32)

uint16_t frameLength

Frame width (4 - 512)

uint16_t position

Data position in the frame

uint8_t watermark

FIFO trigger level

bool txEmptyZero

Transmit zero when buffer becomes empty or last item

bool pack48

Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)

struct `_i2s_transfer`

#include <fsl_i2s.h> Buffer to transfer from or receive audio data into.

Public Members

uint8_t *data

Pointer to data buffer.

size_t dataSize

Buffer size in bytes.

struct `_i2s_handle`

#include <fsl_i2s.h> Members not to be accessed / modified outside of the driver.

Public Members

volatile uint32_t state

State of transfer

i2s_transfer_callback_t completionCallback

Callback function pointer

`void *userData`
Application data passed to callback

`bool oneChannel`
true mono, false stereo

`uint8_t dataLength`
Data length (4 - 32)

`bool pack48`
Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)

`uint8_t watermark`
FIFO trigger level

`bool useFifo48H`
When dataLength 17-24: true use FIFOWR48H, false use FIFOWR

`volatile i2s_transfer_t i2sQueue[(4U)]`
Transfer queue storing transfer buffers

`volatile uint8_t queueUser`
Queue index where user's next transfer will be stored

`volatile uint8_t queueDriver`
Queue index of buffer actually used by the driver

`volatile uint32_t errorCount`
Number of buffer underruns/overruns

`volatile uint32_t transferCount`
Number of bytes transferred

2.18 IAP: In Application Programming Driver

`status_t IAP_ReadPartID(uint32_t *partID)`

Read part identification number.

This function is used to read the part identification number.

Parameters

- `partID` – Address to store the part identification number.

Return values

`kStatus_IAP_Success` – Api has been executed successfully.

`status_t IAP_ReadBootCodeVersion(uint32_t *bootCodeVersion)`

Read boot code version number.

This function is used to read the boot code version number.

note Boot code version is two 32-bit words. Word 0 is the major version, word 1 is the minor version.

Parameters

- `bootCodeVersion` – Address to store the boot code version.

Return values

`kStatus_IAP_Success` – Api has been executed successfully.

```
void IAP_ReinvokeISP(uint8_t ispType, uint32_t *status)
```

Reinvoke ISP.

This function is used to invoke the boot loader in ISP mode. It maps boot vectors and configures the peripherals for ISP.

note The error response will be returned when IAP is disabled or an invalid ISP type selection appears. The call won't return unless an error occurs, so there can be no status code.

Parameters

- `ispType` – ISP type selection.
- `status` – store the possible status.

Return values

`kStatus_IAP_ReinvokeISPConfig` – reinvoke configuration error.

```
status_t IAP_ReadUniqueID(uint32_t *uniqueID)
```

Read unique identification.

This function is used to read the unique id.

Parameters

- `uniqueID` – store the uniqueID.

Return values

`kStatus_IAP_Success` – Api has been executed successfully.

```
status_t IAP_PrepareSectorForWrite(uint32_t startSector, uint32_t endSector)
```

Prepare sector for write operation.

This function prepares sector(s) for write/erase operation. This function must be called before calling the `IAP_CopyRamToFlash()` or `IAP_EraseSector()` or `IAP_ErasePage()` function. The end sector number must be greater than or equal to the start sector number.

Parameters

- `startSector` – Start sector number.
- `endSector` – End sector number.

Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_InvalidSector` – Sector number is invalid or end sector number is greater than start sector number.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

```
status_t IAP_CopyRamToFlash(uint32_t dstAddr, uint32_t *srcAddr, uint32_t numOfBytes,
                           uint32_t systemCoreClock)
```

Copy RAM to flash.

This function programs the flash memory. Corresponding sectors must be prepared via `IAP_PrepareSectorForWrite` before calling this function.

Parameters

- `dstAddr` – Destination flash address where data bytes are to be written, the address should be multiples of `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES` boundary.

- `srcAddr` – Source ram address from where data bytes are to be read.
- `numOfBytes` – Number of bytes to be written, it should be multiples of `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES`, and ranges from `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES` to `FSL_FEATURE_SYSCON_FLASH_SECTOR_SIZE_BYTES`.
- `systemCoreClock` – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_SrcAddrError` – Source address is not on word boundary.
- `kStatus_IAP_DstAddrError` – Destination address is not on a correct boundary.
- `kStatus_IAP_SrcAddrNotMapped` – Source address is not mapped in the memory map.
- `kStatus_IAP_DstAddrNotMapped` – Destination address is not mapped in the memory map.
- `kStatus_IAP_CountError` – Byte count is not multiple of 4 or is not a permitted value.
- `kStatus_IAP_NotPrepared` – Command to prepare sector for write operation has not been executed.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

status_t IAP_EraseSector(uint32_t startSector, uint32_t endSector, uint32_t systemCoreClock)

Erase sector.

This function erases sector(s). The end sector number must be greater than or equal to the start sector number.

Parameters

- `startSector` – Start sector number.
- `endSector` – End sector number.
- `systemCoreClock` – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_InvalidSector` – Sector number is invalid or end sector number is greater than start sector number.
- `kStatus_IAP_NotPrepared` – Command to prepare sector for write operation has not been executed.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

status_t IAP_ErasePage(uint32_t startPage, uint32_t endPage, uint32_t systemCoreClock)

Erase page.

This function erases page(s). The end page number must be greater than or equal to the start page number.

Parameters

- startPage – Start page number.
- endPage – End page number.
- systemCoreClock – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

Return values

- kStatus_IAP_Success – Api has been executed successfully.
- kStatus_IAP_NoPower – Flash memory block is powered down.
- kStatus_IAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_IAP_InvalidSector – Page number is invalid or end page number is greater than start page number.
- kStatus_IAP_NotPrepared – Command to prepare sector for write operation has not been executed.
- kStatus_IAP_Busy – Flash programming hardware interface is busy.

status_t IAP_BlankCheckSector(uint32_t startSector, uint32_t endSector)

Blank check sector(s)

Blank check single or multiples sectors of flash memory. The end sector number must be greater than or equal to the start sector number. It can be used to verify the sector erasure after IAP_EraseSector call.

Parameters

- startSector – Start sector number.
- endSector – End sector number.

Return values

- kStatus_IAP_Success – One or more sectors are in erased state.
- kStatus_IAP_NoPower – Flash memory block is powered down.
- kStatus_IAP_NoClock – Flash memory block or controller is not clocked.
- kStatus_IAP_SectorNotblank – One or more sectors are not blank.

status_t IAP_Compare(uint32_t dstAddr, uint32_t *srcAddr, uint32_t numOfBytes)

Compare memory contents of flash with ram.

This function compares the contents of flash and ram. It can be used to verify the flash memory contents after IAP_CopyRamToFlash call.

Parameters

- dstAddr – Destination flash address.
- srcAddr – Source ram address.
- numOfBytes – Number of bytes to be compared.

Return values

- kStatus_IAP_Success – Contents of flash and ram match.

- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_AddrError` – Address is not on word boundary.
- `kStatus_IAP_AddrNotMapped` – Address is not mapped in the memory map.
- `kStatus_IAP_CountError` – Byte count is not multiple of 4 or is not a permitted value.
- `kStatus_IAP_CompareError` – Destination and source memory contents do not match.

`status_t IAP_ExtendedFlashSignatureRead(uint32_t startPage, uint32_t endPage, uint32_t numOfStates, uint32_t *signature)`

Extended Read signature.

This function calculates the signature value for one or more pages of on-chip flash memory.

Parameters

- `startPage` – Start page number.
- `endPage` – End page number.
- `numOfStates` – Number of wait states.
- `signature` – Address to store the signature value.

Return values

`kStatus_IAP_Success` – Api has been executed successfully.

`status_t IAP_ReadFlashSignature(uint32_t *signature)`

Read flash signature.

This function is used to obtain a 32-bit signature value of the entire flash memory.

Parameters

- `signature` – Address to store the 32-bit generated signature value.

Return values

`kStatus_IAP_Success` – Api has been executed successfully.

`FSL_IAP_DRIVER_VERSION`

iap status codes.

Values:

enumerator `kStatus_IAP_Success`

Api is executed successfully

enumerator `kStatus_IAP_InvalidCommand`

Invalid command

enumerator `kStatus_IAP_SrcAddrError`

Source address is not on word boundary

enumerator `kStatus_IAP_DstAddrError`

Destination address is not on a correct boundary

enumerator `kStatus_IAP_SrcAddrNotMapped`

Source address is not mapped in the memory map

enumerator kStatus_IAP_DstAddrNotMapped

Destination address is not mapped in the memory map

enumerator kStatus_IAP_CountError

Byte count is not multiple of 4 or is not a permitted value

enumerator kStatus_IAP_InvalidSector

Sector/page number is invalid or end sector/page number is greater than start sector/page number

enumerator kStatus_IAP_SectorNotblank

One or more sectors are not blank

enumerator kStatus_IAP_NotPrepared

Command to prepare sector for write operation has not been executed

enumerator kStatus_IAP_CompareError

Destination and source memory contents do not match

enumerator kStatus_IAP_Busy

Flash programming hardware interface is busy

enumerator kStatus_IAP_ParamError

Insufficient number of parameters or invalid parameter

enumerator kStatus_IAP_AddrError

Address is not on word boundary

enumerator kStatus_IAP_AddrNotMapped

Address is not mapped in the memory map

enumerator kStatus_IAP_NoPower

Flash memory block is powered down

enumerator kStatus_IAP_NoClock

Flash memory block or controller is not clocked

enumerator kStatus_IAP_ReinvokeISPConfig

Reinvoke configuration error

enum _iap_commands

iap command codes.

Values:

enumerator kIapCmd_IAP_ReadFactorySettings

Read the factory settings

enumerator kIapCmd_IAP_PrepareSectorforWrite

Prepare Sector for write

enumerator kIapCmd_IAP_CopyRamToFlash

Copy RAM to flash

enumerator kIapCmd_IAP_EraseSector

Erase Sector

enumerator kIapCmd_IAP_BlankCheckSector

Blank check sector

enumerator kIapCmd_IAP_ReadPartId

Read part id

enumerator kIapCmd_IAP_Read_BootromVersion
Read bootrom version

enumerator kIapCmd_IAP_Compare
Compare

enumerator kIapCmd_IAP_ReinvokeISP
Reinvoke ISP

enumerator kIapCmd_IAP_ReadUid
Read Uid

enumerator kIapCmd_IAP_ErasePage
Erase Page

enumerator kIapCmd_IAP_ReadSignature
Read Signature

enumerator kIapCmd_IAP_ExtendedReadSignature
Extended Read Signature

enumerator kIapCmd_IAP_ReadEEPROMPage
Read EEPROM page

enumerator kIapCmd_IAP_WriteEEPROMPage
Write EEPROM page

enum _flash_access_time
Flash memory access time.

Values:

enumerator kFlash_IAP_OneSystemClockTime

enumerator kFlash_IAP_TwoSystemClockTime
1 system clock flash access time

enumerator kFlash_IAP_ThreeSystemClockTime
2 system clock flash access time

2.19 INPUTMUX: Input Multiplexing Driver

FSL_INPUTMUX_DRIVER_VERSION
Group interrupt driver version for SDK.

enum _inputmux_connection_t
INPUTMUX connections type.

Values:

enumerator kINPUTMUX_MainOscToFreqmeas
Frequency measure.

enumerator kINPUTMUX_Fro12MhzToFreqmeas

enumerator kINPUTMUX_WdtOscToFreqmeas

enumerator kINPUTMUX_32KhzOscToFreqmeas

enumerator kINPUTMUX_MainClkToFreqmeas

enumerator kINPUTMUX_GpioPort0Pin4ToFreqmeas
enumerator kINPUTMUX_GpioPort0Pin20ToFreqmeas
enumerator kINPUTMUX_GpioPort0Pin24ToFreqmeas
enumerator kINPUTMUX_GpioPort1Pin4ToFreqmeas
Pin Interrupt.
enumerator kINPUTMUX_GpioPort0Pin0ToPintsel
enumerator kINPUTMUX_GpioPort0Pin1ToPintsel
enumerator kINPUTMUX_GpioPort0Pin2ToPintsel
enumerator kINPUTMUX_GpioPort0Pin3ToPintsel
enumerator kINPUTMUX_GpioPort0Pin4ToPintsel
enumerator kINPUTMUX_GpioPort0Pin5ToPintsel
enumerator kINPUTMUX_GpioPort0Pin6ToPintsel
enumerator kINPUTMUX_GpioPort0Pin7ToPintsel
enumerator kINPUTMUX_GpioPort0Pin8ToPintsel
enumerator kINPUTMUX_GpioPort0Pin9ToPintsel
enumerator kINPUTMUX_GpioPort0Pin10ToPintsel
enumerator kINPUTMUX_GpioPort0Pin11ToPintsel
enumerator kINPUTMUX_GpioPort0Pin12ToPintsel
enumerator kINPUTMUX_GpioPort0Pin13ToPintsel
enumerator kINPUTMUX_GpioPort0Pin14ToPintsel
enumerator kINPUTMUX_GpioPort0Pin15ToPintsel
enumerator kINPUTMUX_GpioPort0Pin16ToPintsel
enumerator kINPUTMUX_GpioPort0Pin17ToPintsel
enumerator kINPUTMUX_GpioPort0Pin18ToPintsel
enumerator kINPUTMUX_GpioPort0Pin19ToPintsel
enumerator kINPUTMUX_GpioPort0Pin20ToPintsel
enumerator kINPUTMUX_GpioPort0Pin21ToPintsel
enumerator kINPUTMUX_GpioPort0Pin22ToPintsel
enumerator kINPUTMUX_GpioPort0Pin23ToPintsel
enumerator kINPUTMUX_GpioPort0Pin24ToPintsel
enumerator kINPUTMUX_GpioPort0Pin25ToPintsel
enumerator kINPUTMUX_GpioPort0Pin26ToPintsel
enumerator kINPUTMUX_GpioPort0Pin27ToPintsel

enumerator kINPUTMUX_GpioPort0Pin28ToPintsel
enumerator kINPUTMUX_GpioPort0Pin29ToPintsel
enumerator kINPUTMUX_GpioPort0Pin30ToPintsel
enumerator kINPUTMUX_GpioPort0Pin31ToPintsel
enumerator kINPUTMUX_GpioPort1Pin0ToPintsel
enumerator kINPUTMUX_GpioPort1Pin1ToPintsel
enumerator kINPUTMUX_GpioPort1Pin2ToPintsel
enumerator kINPUTMUX_GpioPort1Pin3ToPintsel
enumerator kINPUTMUX_GpioPort1Pin4ToPintsel
enumerator kINPUTMUX_GpioPort1Pin5ToPintsel
enumerator kINPUTMUX_GpioPort1Pin6ToPintsel
enumerator kINPUTMUX_GpioPort1Pin7ToPintsel
enumerator kINPUTMUX_GpioPort1Pin8ToPintsel
enumerator kINPUTMUX_GpioPort1Pin9ToPintsel
enumerator kINPUTMUX_GpioPort1Pin10ToPintsel
enumerator kINPUTMUX_GpioPort1Pin11ToPintsel
enumerator kINPUTMUX_GpioPort1Pin12ToPintsel
enumerator kINPUTMUX_GpioPort1Pin13ToPintsel
enumerator kINPUTMUX_GpioPort1Pin14ToPintsel
enumerator kINPUTMUX_GpioPort1Pin15ToPintsel
enumerator kINPUTMUX_GpioPort1Pin16ToPintsel
enumerator kINPUTMUX_GpioPort1Pin17ToPintsel
enumerator kINPUTMUX_GpioPort1Pin18ToPintsel
enumerator kINPUTMUX_GpioPort1Pin19ToPintsel
enumerator kINPUTMUX_GpioPort1Pin20ToPintsel
enumerator kINPUTMUX_GpioPort1Pin21ToPintsel
enumerator kINPUTMUX_GpioPort1Pin22ToPintsel
enumerator kINPUTMUX_GpioPort1Pin23ToPintsel
enumerator kINPUTMUX_GpioPort1Pin24ToPintsel
enumerator kINPUTMUX_GpioPort1Pin25ToPintsel
enumerator kINPUTMUX_GpioPort1Pin26ToPintsel
enumerator kINPUTMUX_GpioPort1Pin27ToPintsel
enumerator kINPUTMUX_GpioPort1Pin28ToPintsel

enumerator kINPUTMUX_GpioPort1Pin29ToPintsel

enumerator kINPUTMUX_GpioPort1Pin30ToPintsel

enumerator kINPUTMUX_GpioPort1Pin31ToPintsel
DMA ITRIG.

enumerator kINPUTMUX_Adc0SeqaIrqToDma

enumerator kINPUTMUX_ADC0SeqbIrqToDma

enumerator kINPUTMUX_Sct0DmaReq0ToDma

enumerator kINPUTMUX_Sct0DmaReq1ToDma

enumerator kINPUTMUX_Ctimer0M0ToDma

enumerator kINPUTMUX_Ctimer0M1ToDma

enumerator kINPUTMUX_Ctimer1M0ToDma

enumerator kINPUTMUX_Ctimer3M0ToDma

enumerator kINPUTMUX_PinInt0ToDma

enumerator kINPUTMUX_PinInt1ToDma

enumerator kINPUTMUX_PinInt2ToDma

enumerator kINPUTMUX_PinInt3ToDma

enumerator kINPUTMUX_Otrig0ToDma

enumerator kINPUTMUX_Otrig1ToDma

enumerator kINPUTMUX_Otrig2ToDma

enumerator kINPUTMUX_Otrig3ToDma
DMA OTRIG.

enumerator kINPUTMUX_DmaFlexcomm0RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm0TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm1RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm1TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm2RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm2TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm3RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm3TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm4RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm4TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm5RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm5TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm6RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm6TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm7RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm7TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaChannel18_TrigoutToTriginChannels

enumerator kINPUTMUX_DmaChannel19_TrigoutToTriginChannels

typedef enum *inputmux_connection_t* inputmux_connection_t
INPUTMUX connections type.

void INPUTMUX_Init(void *base)
Initialize INPUTMUX peripheral.

This function enables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

void INPUTMUX_AttachSignal(void *base, uint32_t index, *inputmux_connection_t* connection)
Attaches a signal.

This function attaches multiplexed signals from INPUTMUX to target signals. For example, to attach GPIO PORT0 Pin 5 to PINT peripheral, do the following:

```
INPUTMUX_AttachSignal(INPUTMUX, 2, kINPUTMUX_GpioPort0Pin5ToPintsel);
```

In this example, INTMUX has 8 registers for PINT, PINT_SEL0~PINT_SEL7. With parameter index specified as 2, this function configures register PINT_SEL2.

Parameters

- base – Base address of the INPUTMUX peripheral.
- index – The serial number of destination register in the group of INPUTMUX registers with same name.
- connection – Applies signal from source signals collection to target signal.

Return values

None. –

void INPUTMUX_Deinit(void *base)
Deinitialize INPUTMUX peripheral.

This function disables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

PINTSEL_PMUX_ID

Periphinmux IDs.

DMA_TRIG0_PMUX_ID

DMA_OTRIG_PMUX_ID

FREQMEAS_PMUX_ID

PMUX_SHIFT

2.20 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_SIZEALIGN(var, alignbytes)

Macro to define a variable with L1 d-cache line size alignment

Macro to define a variable with L2 cache line size alignment

Macro to change a value to a given size aligned value

AT_NONCACHEABLE_SECTION(var)

Define a variable *var*, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_NONCACHEABLE_SECTION_INIT(var)

Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

enum _status_groups

Status group numbers.

Values:

enumerator kStatusGroup_Generic

Group number for generic status codes.

enumerator kStatusGroup_FLASH

Group number for FLASH status codes.

enumerator kStatusGroup_LPSPi
Group number for LPSPi status codes.

enumerator kStatusGroup_FLEXIO_SPI
Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI
Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
Group number for LPI2C status codes.

enumerator kStatusGroup_UART
Group number for UART status codes.

enumerator kStatusGroup_I2C
Group number for I2C status codes.

enumerator kStatusGroup_LPSCI
Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
Group number for LPUART status codes.

enumerator kStatusGroup_SPI
Group number for SPI status code.

enumerator kStatusGroup_XRDC
Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPi
Group number for ECSPi status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator `kStatusGroup_LPC_SPI_SSP`
Group number for LPC_SPI_SSP status codes.

enumerator `kStatusGroup_I3C`
Group number for I3C status codes

enumerator `kStatusGroup_LPC_I2C_1`
Group number for LPC_I2C_1 status codes.

enumerator `kStatusGroup_NOTIFIER`
Group number for NOTIFIER status codes.

enumerator `kStatusGroup_DebugConsole`
Group number for debug console status codes.

enumerator `kStatusGroup_SEMC`
Group number for SEMC status codes.

enumerator `kStatusGroup_ApplicationRangeStart`
Starting number for application groups.

enumerator `kStatusGroup_IAP`
Group number for IAP status codes

enumerator `kStatusGroup_SFA`
Group number for SFA status codes

enumerator `kStatusGroup_SPC`
Group number for SPC status codes.

enumerator `kStatusGroup_PUF`
Group number for PUF status codes.

enumerator `kStatusGroup_TOUCH_PANEL`
Group number for touch panel status codes

enumerator `kStatusGroup_VBAT`
Group number for VBAT status codes

enumerator `kStatusGroup_XSPI`
Group number for XSPI status codes

enumerator `kStatusGroup_PNGDEC`
Group number for PNGDEC status codes

enumerator `kStatusGroup_JPEGDEC`
Group number for JPEGDEC status codes

enumerator `kStatusGroup_HAL_GPIO`
Group number for HAL GPIO status codes.

enumerator `kStatusGroup_HAL_UART`
Group number for HAL UART status codes.

enumerator `kStatusGroup_HAL_TIMER`
Group number for HAL TIMER status codes.

enumerator `kStatusGroup_HAL_SPI`
Group number for HAL SPI status codes.

enumerator `kStatusGroup_HAL_I2C`
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

enumerator kStatusGroup_OSA
Group number for OSA status codes.

enumerator kStatusGroup_COMMON_TASK
Group number for Common task status codes.

enumerator kStatusGroup_MSG
Group number for messaging status codes.

enumerator kStatusGroup_SDK_OCOTP
Group number for OCOTP status codes.

enumerator kStatusGroup_SDK_FLEXSPINOR
Group number for FLEXSPINOR status codes.

enumerator kStatusGroup_CODEC
Group number for codec status codes.

enumerator kStatusGroup_ASRC
Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
Group number for codec status codes.

enumerator `kStatusGroup_SDIO_SLV`
Group number for SDIO_SLV status codes.

enumerator `kStatusGroup_MECC`
Group number for MECC status codes.

enumerator `kStatusGroup_ENET_QOS`
Group number for ENET_QOS status codes.

enumerator `kStatusGroup_LOG`
Group number for LOG status codes.

enumerator `kStatusGroup_I3CBUS`
Group number for I3CBUS status codes.

enumerator `kStatusGroup_QSCI`
Group number for QSCI status codes.

enumerator `kStatusGroup_ELEMU`
Group number for ELEMU status codes.

enumerator `kStatusGroup_QUEUEDSPI`
Group number for QSPI status codes.

enumerator `kStatusGroup_POWER_MANAGER`
Group number for POWER_MANAGER status codes.

enumerator `kStatusGroup_IPED`
Group number for IPED status codes.

enumerator `kStatusGroup_ELS_PKC`
Group number for ELS PKC status codes.

enumerator `kStatusGroup_CSS_PKC`
Group number for CSS PKC status codes.

enumerator `kStatusGroup_HOSTIF`
Group number for HOSTIF status codes.

enumerator `kStatusGroup_CLIF`
Group number for CLIF status codes.

enumerator `kStatusGroup_BMA`
Group number for BMA status codes.

enumerator `kStatusGroup_NETC`
Group number for NETC status codes.

enumerator `kStatusGroup_ELE`
Group number for ELE status codes.

enumerator `kStatusGroup_GLIKEY`
Group number for GLIKEY status codes.

enumerator `kStatusGroup_AON_POWER`
Group number for AON_POWER status codes.

enumerator `kStatusGroup_AON_COMMON`
Group number for AON_COMMON status codes.

enumerator `kStatusGroup_ENDAT3`
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE

Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX

Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC

Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT

Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT

Group number for A-format status codes.

Generic status return codes.

Values:

enumerator kStatus_Success

Generic status for Success.

enumerator kStatus_Fail

Generic status for Fail.

enumerator kStatus_ReadOnly

Generic status for read only failure.

enumerator kStatus_OutOfRange

Generic status for out of range access.

enumerator kStatus_InvalidArgument

Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

`void SDK_Free(void *ptr)`

Free memory.

Parameters

- `ptr` – The memory to be release.

`void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)`

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- `delayTime_us` – Delay time in unit of microsecond.
- `coreClock_Hz` – Core clock frequency with Hz.

`static inline status_t EnableIRQ(IRQn_Type interrupt)`

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt enabled successfully
- `kStatus_Fail` – Failed to enable the interrupt

`static inline status_t DisableIRQ(IRQn_Type interrupt)`

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt disabled successfully
- `kStatus_Fail` – Failed to disable the interrupt

`static inline status_t EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)`

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management

mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

void EnableDeepSleepIRQ(IRQn_Type interrupt)

Enable specific interrupt for wake-up from deep-sleep mode.

Enable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

Note: This function also enables the interrupt in the NVIC (EnableIRQ() is called internally).

Parameters

- interrupt – The IRQ number.

void DisableDeepSleepIRQ(IRQn_Type interrupt)

Disable specific interrupt for wake-up from deep-sleep mode.

Disable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

Note: This function also disables the interrupt in the NVIC (DisableIRQ() is called internally).

Parameters

- interrupt – The IRQ number.

static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t newValue)

static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version		Minor Version		Bug Fix		
31	25	24	17	16	9	8	0

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING();” need to be added at the end of each case section which misses “break;”statement.

MSDK_REG_SECURE_ADDR(x)

Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

Convert the register address to the one used in non-secure mode.

MSDK_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.21 ADC: 12-bit SAR Analog-to-Digital Converter Driver

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

Initialize the ADC module.

Parameters

- base – ADC peripheral base address.
- config – Pointer to configuration structure, see to *adc_config_t*.

void ADC_Deinit(ADC_Type *base)

Deinitialize the ADC module.

Parameters

- base – ADC peripheral base address.

void ADC_GetDefaultConfig(*adc_config_t* *config)

Gets an available pre-defined settings for initial configuration.

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

```
config->clockMode = kADC_ClockSynchronousMode;
config->clockDividerNumber = 0U;
config->resolution = kADC_Resolution12bit;
config->enableBypassCalibration = false;
config->sampleTimeNumber = 0U;
config->extendSampleTimeNumber = kADC_ExtendSampleTimeNotUsed;
```

Parameters

- config – Pointer to configuration structure.

bool ADC_DoSelfCalibration(ADC_Type *base)

Do the hardware self-calibration.

Deprecated:

Do not use this function. It has been superceded by `ADC_DoOffsetCalibration`.

To calibrate the ADC, set the ADC clock to 500 kHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

- `base` – ADC peripheral base address.

Return values

- `true` – Calibration succeed.
- `false` – Calibration failed.

```
bool ADC_DoOffsetCalibration(ADC_Type *base, uint32_t frequency)
```

Do the hardware offset-calibration.

To calibrate the ADC, set the ADC clock to no more than 30 MHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

- `base` – ADC peripheral base address.
- `frequency` – The clock frequency that ADC operates at.

Return values

- `true` – Calibration succeed.
- `false` – Calibration failed.

```
static inline void ADC_EnableConvSeqA(ADC_Type *base, bool enable)
```

Enable the conversion sequence A.

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

- `base` – ADC peripheral base address.
- `enable` – Switcher to enable the feature or not.

```
void ADC_SetConvSeqAConfig(ADC_Type *base, const adc_conv_seq_config_t *config)
```

Configure the conversion sequence A.

Parameters

- `base` – ADC peripheral base address.
- `config` – Pointer to configuration structure, see to `adc_conv_seq_config_t`.

```
static inline void ADC_DoSoftwareTriggerConvSeqA(ADC_Type *base)
```

Do trigger the sequence's conversion by software.

Parameters

- `base` – ADC peripheral base address.

```
static inline void ADC_EnableConvSeqABurstMode(ADC_Type *base, bool enable)
```

Enable the burst conversion of sequence A.

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

- base – ADC peripheral base address.
- enable – Switcher to enable this feature.

```
static inline void ADC_SetConvSeqAHighPriority(ADC_Type *base)
```

Set the high priority for conversion sequence A.

Parameters

- base – ADC peripheral base address.

```
static inline void ADC_EnableConvSeqB(ADC_Type *base, bool enable)
```

Enable the conversion sequence B.

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. When the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

- base – ADC peripheral base address.
- enable – Switcher to enable the feature or not.

```
void ADC_SetConvSeqBConfig(ADC_Type *base, const adc_conv_seq_config_t *config)
```

Configure the conversion sequence B.

Parameters

- base – ADC peripheral base address.
- config – Pointer to configuration structure, see to *adc_conv_seq_config_t*.

```
static inline void ADC_DoSoftwareTriggerConvSeqB(ADC_Type *base)
```

Do trigger the sequence's conversion by software.

Parameters

- base – ADC peripheral base address.

```
static inline void ADC_EnableConvSeqBBurstMode(ADC_Type *base, bool enable)
```

Enable the burst conversion of sequence B.

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

- base – ADC peripheral base address.
- enable – Switcher to enable this feature.

```
static inline void ADC_SetConvSeqBHighPriority(ADC_Type *base)
```

Set the high priority for conversion sequence B.

Parameters

- base – ADC peripheral base address.

```
bool ADC_GetConvSeqAGlobalConversionResult(ADC_Type *base, adc_result_info_t *info)
```

Get the global ADC conversion information of sequence A.

Parameters

- base – ADC peripheral base address.
- info – Pointer to information structure, see to `adc_result_info_t`;

Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
bool ADC_GetConvSeqBGlobalConversionResult(ADC_Type *base, adc_result_info_t *info)
```

Get the global ADC conversion information of sequence B.

Parameters

- base – ADC peripheral base address.
- info – Pointer to information structure, see to `adc_result_info_t`;

Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
bool ADC_GetChannelConversionResult(ADC_Type *base, uint32_t channel, adc_result_info_t *info)
```

Get the channel's ADC conversion completed under each conversion sequence.

Parameters

- base – ADC peripheral base address.
- channel – The indicated channel number.
- info – Pointer to information structure, see to `adc_result_info_t`;

Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
static inline void ADC_SetThresholdPair0(ADC_Type *base, uint32_t lowValue, uint32_t highValue)
```

Set the threshold pair 0 with low and high value.

Parameters

- base – ADC peripheral base address.
- lowValue – LOW threshold value.
- highValue – HIGH threshold value.

```
static inline void ADC_SetThresholdPair1(ADC_Type *base, uint32_t lowValue, uint32_t highValue)
```

Set the threshold pair 1 with low and high value.

Parameters

- `base` – ADC peripheral base address.
- `lowValue` – LOW threshold value. The available value is with 12-bit.
- `highValue` – HIGH threshold value. The available value is with 12-bit.

static inline void ADC_SetChannelWithThresholdPair0(ADC_Type *base, uint32_t channelMask)
Set given channels to apply the threshold pare 0.

Parameters

- `base` – ADC peripheral base address.
- `channelMask` – Indicated channels' mask.

static inline void ADC_SetChannelWithThresholdPair1(ADC_Type *base, uint32_t channelMask)
Set given channels to apply the threshold pare 1.

Parameters

- `base` – ADC peripheral base address.
- `channelMask` – Indicated channels' mask.

static inline void ADC_EnableInterrupts(ADC_Type *base, uint32_t mask)
Enable interrupts for conversion sequences.

Parameters

- `base` – ADC peripheral base address.
- `mask` – Mask of interrupt mask value for global block except each channel, see to `_adc_interrupt_enable`.

static inline void ADC_DisableInterrupts(ADC_Type *base, uint32_t mask)
Disable interrupts for conversion sequence.

Parameters

- `base` – ADC peripheral base address.
- `mask` – Mask of interrupt mask value for global block except each channel, see to `_adc_interrupt_enable`.

static inline void ADC_EnableThresholdCompareInterrupt(ADC_Type *base, uint32_t channel,
adc_threshold_interrupt_mode_t mode)
Enable the interrupt of threshold compare event for each channel.

Parameters

- `base` – ADC peripheral base address.
- `channel` – Channel number.
- `mode` – Interrupt mode for threshold compare event, see to `adc_threshold_interrupt_mode_t`.

static inline uint32_t ADC_GetStatusFlags(ADC_Type *base)
Get status flags of ADC module.

Parameters

- `base` – ADC peripheral base address.

Returns

Mask of status flags of module, see to `_adc_status_flags`.

static inline void ADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clear status flags of ADC module.

Parameters

- base – ADC peripheral base address.
- mask – Mask of status flags of module, see to `_adc_status_flags`.

FSL_ADC_DRIVER_VERSION

ADC driver version 2.6.0.

enum _adc_status_flags

Flags.

Values:

enumerator kADC_ThresholdCompareFlagOnChn0

Threshold comparison event on Channel 0.

enumerator kADC_ThresholdCompareFlagOnChn1

Threshold comparison event on Channel 1.

enumerator kADC_ThresholdCompareFlagOnChn2

Threshold comparison event on Channel 2.

enumerator kADC_ThresholdCompareFlagOnChn3

Threshold comparison event on Channel 3.

enumerator kADC_ThresholdCompareFlagOnChn4

Threshold comparison event on Channel 4.

enumerator kADC_ThresholdCompareFlagOnChn5

Threshold comparison event on Channel 5.

enumerator kADC_ThresholdCompareFlagOnChn6

Threshold comparison event on Channel 6.

enumerator kADC_ThresholdCompareFlagOnChn7

Threshold comparison event on Channel 7.

enumerator kADC_ThresholdCompareFlagOnChn8

Threshold comparison event on Channel 8.

enumerator kADC_ThresholdCompareFlagOnChn9

Threshold comparison event on Channel 9.

enumerator kADC_ThresholdCompareFlagOnChn10

Threshold comparison event on Channel 10.

enumerator kADC_ThresholdCompareFlagOnChn11

Threshold comparison event on Channel 11.

enumerator kADC_OverrunFlagForChn0

Mirror the OVERRUN status flag from the result register for ADC channel 0.

enumerator kADC_OverrunFlagForChn1

Mirror the OVERRUN status flag from the result register for ADC channel 1.

enumerator kADC_OverrunFlagForChn2

Mirror the OVERRUN status flag from the result register for ADC channel 2.

enumerator kADC_OverrunFlagForChn3

Mirror the OVERRUN status flag from the result register for ADC channel 3.

enumerator kADC_OverrunFlagForChn4

Mirror the OVERRUN status flag from the result register for ADC channel 4.

enumerator kADC_OverrunFlagForChn5

Mirror the OVERRUN status flag from the result register for ADC channel 5.

enumerator kADC_OverrunFlagForChn6

Mirror the OVERRUN status flag from the result register for ADC channel 6.

enumerator kADC_OverrunFlagForChn7

Mirror the OVERRUN status flag from the result register for ADC channel 7.

enumerator kADC_OverrunFlagForChn8

Mirror the OVERRUN status flag from the result register for ADC channel 8.

enumerator kADC_OverrunFlagForChn9

Mirror the OVERRUN status flag from the result register for ADC channel 9.

enumerator kADC_OverrunFlagForChn10

Mirror the OVERRUN status flag from the result register for ADC channel 10.

enumerator kADC_OverrunFlagForChn11

Mirror the OVERRUN status flag from the result register for ADC channel 11.

enumerator kADC_GlobalOverrunFlagForSeqA

Mirror the global OVERRUN status flag for conversion sequence A.

enumerator kADC_GlobalOverrunFlagForSeqB

Mirror the global OVERRUN status flag for conversion sequence B.

enumerator kADC_ConvSeqAInterruptFlag

Sequence A interrupt/DMA trigger.

enumerator kADC_ConvSeqBInterruptFlag

Sequence B interrupt/DMA trigger.

enumerator kADC_ThresholdCompareInterruptFlag

Threshold comparison interrupt flag.

enumerator kADC_OverrunInterruptFlag

Overrun interrupt flag.

enum _adc_interrupt_enable

Interrupts.

Note: Not all the interrupt options are listed here

Values:

enumerator kADC_ConvSeqAInterruptEnable

Enable interrupt upon completion of each individual conversion in sequence A, or entire sequence.

enumerator kADC_ConvSeqBInterruptEnable

Enable interrupt upon completion of each individual conversion in sequence B, or entire sequence.

enumerator kADC_OverrunInterruptEnable

Enable the detection of an overrun condition on any of the channel data registers will cause an overrun interrupt/DMA trigger.

enum _adc_clock_mode

Define selection of clock mode.

Values:

enumerator kADC_ClockSynchronousMode

The ADC clock would be derived from the system clock based on “clockDividerNumber”.

enumerator kADC_ClockAsynchronousMode

The ADC clock would be based on the SYSCON block’s divider.

enum _adc_resolution

Define selection of resolution.

Values:

enumerator kADC_Resolution6bit

6-bit resolution.

enumerator kADC_Resolution8bit

8-bit resolution.

enumerator kADC_Resolution10bit

10-bit resolution.

enumerator kADC_Resolution12bit

12-bit resolution.

enum _adc_voltage_range

Define range of the analog supply voltage VDDA.

Values:

enumerator kADC_HighVoltageRange

enumerator kADC_LowVoltageRange

enum _adc_trigger_polarity

Define selection of polarity of selected input trigger for conversion sequence.

Values:

enumerator kADC_TriggerPolarityNegativeEdge

A negative edge launches the conversion sequence on the trigger(s).

enumerator kADC_TriggerPolarityPositiveEdge

A positive edge launches the conversion sequence on the trigger(s).

enum _adc_priority

Define selection of conversion sequence’s priority.

Values:

enumerator kADC_PriorityLow

This sequence would be preempted when another sequence is started.

enumerator kADC_PriorityHigh

This sequence would preempt other sequence even when it is started.

enum _adc_seq_interrupt_mode

Define selection of conversion sequence’s interrupt.

Values:

enumerator kADC_InterruptForEachConversion

The sequence interrupt/DMA trigger will be set at the end of each individual ADC conversion inside this conversion sequence.

enumerator kADC_InterruptForEachSequence

The sequence interrupt/DMA trigger will be set when the entire set of this sequence conversions completes.

enum _adc_threshold_compare_status

Define status of threshold compare result.

Values:

enumerator kADC_ThresholdCompareInRange

LOW threshold $\leq$ conversion value $\leq$ HIGH threshold.

enumerator kADC_ThresholdCompareBelowRange

conversion value $<$ LOW threshold.

enumerator kADC_ThresholdCompareAboveRange

conversion value $>$ HIGH threshold.

enum _adc_threshold_crossing_status

Define status of threshold crossing detection result.

Values:

enumerator kADC_ThresholdCrossingNoDetected

No threshold Crossing detected.

enumerator kADC_ThresholdCrossingDownward

Downward Threshold Crossing detected.

enumerator kADC_ThresholdCrossingUpward

Upward Threshold Crossing Detected.

enum _adc_threshold_interrupt_mode

Define interrupt mode for threshold compare event.

Values:

enumerator kADC_ThresholdInterruptDisabled

Threshold comparison interrupt is disabled.

enumerator kADC_ThresholdInterruptOnOutside

Threshold comparison interrupt is enabled on outside threshold.

enumerator kADC_ThresholdInterruptOnCrossing

Threshold comparison interrupt is enabled on crossing threshold.

enum _adc_inforeresultshift

Define the info result mode of different resolution.

Values:

enumerator kADC_Resolution12bitInfoResultShift

Info result shift of Resolution12bit.

enumerator kADC_Resolution10bitInfoResultShift

Info result shift of Resolution10bit.

enumerator kADC_Resolution8bitInfoResultShift

Info result shift of Resolution8bit.

enumerator kADC_Resolution6bitInfoResultShift

Info result shift of Resolution6bit.

enum _adc_tempsensor_common_mode

Define common modes for Temperature sensor.

Values:

enumerator kADC_HighNegativeOffsetAdded

Temperature sensor common mode: high negative offset added.

enumerator kADC_IntermediateNegativeOffsetAdded

Temperature sensor common mode: intermediate negative offset added.

enumerator kADC_NoOffsetAdded

Temperature sensor common mode: no offset added.

enumerator kADC_LowPositiveOffsetAdded

Temperature sensor common mode: low positive offset added.

enum _adc_second_control

Define source impedance modes for GPADC control.

Values:

enumerator kADC_Impedance621Ohm

Extend ADC sampling time according to source impedance 1: 0.621 kOhm.

enumerator kADC_Impedance55kOhm

Extend ADC sampling time according to source impedance 20 (default): 55 kOhm.

enumerator kADC_Impedance87kOhm

Extend ADC sampling time according to source impedance 31: 87 kOhm.

enumerator kADC_NormalFunctionalMode

TEST mode: Normal functional mode.

enumerator kADC_MultiplexeTestMode

TEST mode: Multiplexer test mode.

enumerator kADC_ADCInUnityGainMode

TEST mode: ADC in unity gain mode.

typedef enum _adc_clock_mode adc_clock_mode_t

Define selection of clock mode.

typedef enum _adc_resolution adc_resolution_t

Define selection of resolution.

typedef enum _adc_voltage_range adc_vdda_range_t

Define range of the analog supply voltage VDDA.

typedef enum _adc_trigger_polarity adc_trigger_polarity_t

Define selection of polarity of selected input trigger for conversion sequence.

typedef enum _adc_priority adc_priority_t

Define selection of conversion sequence's priority.

typedef enum _adc_seq_interrupt_mode adc_seq_interrupt_mode_t

Define selection of conversion sequence's interrupt.

typedef enum _adc_threshold_compare_status adc_threshold_compare_status_t

Define status of threshold compare result.

```
typedef enum _adc_threshold_crossing_status adc_threshold_crossing_status_t
```

Define status of threshold crossing detection result.

```
typedef enum _adc_threshold_interrupt_mode adc_threshold_interrupt_mode_t
```

Define interrupt mode for threshold compare event.

```
typedef enum _adc_inforeresultshift adc_inforeresult_t
```

Define the info result mode of different resolution.

```
typedef enum _adc_tempsensor_common_mode adc_tempsensor_common_mode_t
```

Define common modes for Temperature sensor.

```
typedef enum _adc_second_control adc_second_control_t
```

Define source impedance modes for GPADC control.

```
typedef struct _adc_config adc_config_t
```

Define structure for configuring the block.

```
typedef struct _adc_conv_seq_config adc_conv_seq_config_t
```

Define structure for configuring conversion sequence.

```
typedef struct _adc_result_info adc_result_info_t
```

Define structure of keeping conversion result information.

```
struct _adc_config
```

#include <fsl_adc.h> Define structure for configuring the block.

Public Members

```
adc_clock_mode_t clockMode
```

Select the clock mode for ADC converter.

```
uint32_t clockDividerNumber
```

This field is only available when using kADC_ClockSynchronousMode for “clockMode” field. The divider would be plused by 1 based on the value in this field. The available range is in 8 bits.

```
adc_resolution_t resolution
```

Select the conversion bits.

```
bool enableBypassCalibration
```

By default, a calibration cycle must be performed each time the chip is powered-up. Re-calibration may be warranted periodically - especially if operating conditions have changed. To enable this option would avoid the need to calibrate if offset error is not a concern in the application.

```
uint32_t sampleTimeNumber
```

By default, with value as “0U”, the sample period would be 2.5 ADC clocks. Then, to plus the “sampleTimeNumber” value here. The available value range is in 3 bits.

```
bool enableLowPowerMode
```

If disable low-power mode, ADC remains activated even when no conversions are requested. If enable low-power mode, The ADC is automatically powered-down when no conversions are taking place.

```
adc_vdda_range_t voltageRange
```

Configure the ADC for the appropriate operating range of the analog supply voltage VDDA. Failure to set the area correctly causes the ADC to return incorrect conversion results.

```
struct _adc_conv_seq_config
```

#include <fsl_adc.h> Define structure for configuring conversion sequence.

Public Members

`uint32_t channelMask`

Selects which one or more of the ADC channels will be sampled and converted when this sequence is launched. The masked channels would be involved in current conversion sequence, beginning with the lowest-order. The available range is in 12-bit.

`uint32_t triggerMask`

Selects which one or more of the available hardware trigger sources will cause this conversion sequence to be initiated. The available range is 6-bit.

`adc_trigger_polarity_t triggerPolarity`

Select the trigger to launch conversion sequence.

`bool enableSyncBypass`

To enable this feature allows the hardware trigger input to bypass synchronization flip-flop stages and therefore shorten the time between the trigger input signal and the start of a conversion.

`bool enableSingleStep`

When enabling this feature, a trigger will launch a single conversion on the next channel in the sequence instead of the default response of launching an entire sequence of conversions.

`adc_seq_interrupt_mode_t interruptMode`

Select the interrupt/DMA trigger mode.

`struct _adc_result_info`

`#include <fsl_adc.h>` Define structure of keeping conversion result information.

Public Members

`uint32_t result`

Keep the conversion data value.

`adc_threshold_compare_status_t thresholdCompareStatus`

Keep the threshold compare status.

`adc_threshold_crossing_status_t thresholdCorssingStatus`

Keep the threshold crossing status.

`uint32_t channelNumber`

Keep the channel number for this conversion.

`bool overrunFlag`

Keep the status whether the conversion is overrun or not.

2.22 GPIO: General Purpose I/O

`void GPIO_PortInit(GPIO_Type *base, uint32_t port)`

Initializes the GPIO peripheral.

This function ungates the GPIO clock.

Parameters

- `base` – GPIO peripheral base pointer.
- `port` – GPIO port number.

```
void GPIO_PinInit(GPIO_Type *base, uint32_t port, uint32_t pin, const gpio_pin_config_t
                 *config)
```

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or output pin configuration:

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}
```

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number
- config – GPIO pin configuration pointer

```
static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t port, uint32_t pin, uint8_t output)
```

Sets the output level of the one GPIO pin to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

```
static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t port, uint32_t pin)
```

Reads the current input value of the GPIO PIN.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

FSL_GPIO_DRIVER_VERSION

LPC GPIO driver version.

enum `_gpio_pin_direction`

LPC GPIO direction definition.

Values:

enumerator `kGPIO_DigitalInput`

Set current pin as digital input

enumerator `kGPIO_DigitalOutput`

Set current pin as digital output

typedef enum `_gpio_pin_direction` `gpio_pin_direction_t`

LPC GPIO direction definition.

typedef struct `_gpio_pin_config` `gpio_pin_config_t`

The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

static inline void `GPIO_PortSet(GPIO_Type *base, uint32_t port, uint32_t mask)`

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- `base` – GPIO peripheral base pointer(Typically GPIO)
- `port` – GPIO port number
- `mask` – GPIO pin number macro

static inline void `GPIO_PortClear(GPIO_Type *base, uint32_t port, uint32_t mask)`

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- `base` – GPIO peripheral base pointer(Typically GPIO)
- `port` – GPIO port number
- `mask` – GPIO pin number macro

static inline void `GPIO_PortToggle(GPIO_Type *base, uint32_t port, uint32_t mask)`

Reverses current output logic of the multiple GPIO pins.

Parameters

- `base` – GPIO peripheral base pointer(Typically GPIO)
- `port` – GPIO port number
- `mask` – GPIO pin number macro

struct `_gpio_pin_config`

#include <fsl_gpio.h> The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

Public Members

`gpio_pin_direction_t` `pinDirection`

GPIO direction, input or output

`uint8_t` `outputLogic`

Set default output logic, no use in input

2.23 IOCON: I/O pin configuration

FSL_IOCON_DRIVER_VERSION

IOCON driver version.

typedef struct *iocon_group* iocon_group_t

Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format.

__STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t port, uint8_t pin, uint32_t modefunc)

Sets I/O Control pin mux.

Parameters

- base – : The base of IOCON peripheral on the chip
- port – : GPIO port to mux
- pin – : GPIO pin to mux
- modefunc – : OR'ed values of type IOCON_*

Returns

Nothing

__STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base, const iocon_group_t *pinArray, uint32_t arrayLength)

Set all I/O Control pin muxing.

Parameters

- base – : The base of IOCON peripheral on the chip
- pinArray – : Pointer to array of pin mux selections
- arrayLength – : Number of entries in pinArray

Returns

Nothing

FSL_COMPONENT_ID

IOCON_FUNC0

IOCON function and mode selection definitions.

Note: See the User Manual for specific modes and functions supported by the various pins.
Selects pin function 0

IOCON_FUNC1

Selects pin function 1

IOCON_FUNC2

Selects pin function 2

IOCON_FUNC3

Selects pin function 3

IOCON_FUNC4

Selects pin function 4

IOCON_FUNC5

Selects pin function 5

IOCON_FUNC6

Selects pin function 6

IOCON_FUNC7

Selects pin function 7

struct __iocon_group

#include <fsl_iocon.h> Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format.

2.24 MRT: Multi-Rate Timer

void MRT_Init(MRT_Type *base, const *mrt_config_t* *config)

Ungates the MRT clock and configures the peripheral for basic operation.

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

Parameters

- *base* – Multi-Rate timer peripheral base address
- *config* – Pointer to user's MRT config structure. If MRT has MULTITASK bit field in MODCFG register, param config is useless.

void MRT_Deinit(MRT_Type *base)

Gate the MRT clock.

Parameters

- *base* – Multi-Rate timer peripheral base address

static inline void MRT_GetDefaultConfig(*mrt_config_t* *config)

Fill in the MRT config struct with the default settings.

The default values are:

```
config->enableMultiTask = false;
```

Parameters

- *config* – Pointer to user's MRT config structure.

static inline void MRT_SetupChannelMode(MRT_Type *base, *mrt_chnl_t* channel, const *mrt_timer_mode_t* mode)

Sets up an MRT channel mode.

Parameters

- *base* – Multi-Rate timer peripheral base address
- *channel* – Channel that is being configured.
- *mode* – Timer mode to use for the channel.

static inline void MRT_EnableInterrupts(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Enables the MRT interrupt.

Parameters

- *base* – Multi-Rate timer peripheral base address
- *channel* – Timer channel number

- mask – The interrupts to enable. This is a logical OR of members of the enumeration `mrt_interrupt_enable_t`

```
static inline void MRT_DisableInterrupts(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Disables the selected MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `mrt_interrupt_enable_t`

```
static inline uint32_t MRT_GetEnabledInterrupts(MRT_Type *base, mrt_chnl_t channel)
```

Gets the enabled MRT interrupts.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

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

```
static inline uint32_t MRT_GetStatusFlags(MRT_Type *base, mrt_chnl_t channel)
```

Gets the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

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

```
static inline void MRT_ClearStatusFlags(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Clears the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The status flags to clear. This is a logical OR of members of the enumeration `mrt_status_flags_t`

```
void MRT_UpdateTimerPeriod(MRT_Type *base, mrt_chnl_t channel, uint32_t count, bool  
                           immediateLoad)
```

Used to update the timer period in units of count.

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

- `count` – Timer period in units of ticks
- `immediateLoad` – `true`: Load the new value immediately into the TIMER register; `false`: Load the new value at the end of current timer interval

static inline uint32_t MRT_GetCurrentTimerCount(MRT_Type *base, *mrt_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number

Returns

Current timer counting value in ticks

static inline void MRT_StartTimer(MRT_Type *base, *mrt_chnl_t* channel, uint32_t count)

Starts the timer counting.

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.
- `count` – Timer period in units of ticks. Count can contain the LOAD bit, which control the force load feature.

static inline void MRT_StopTimer(MRT_Type *base, *mrt_chnl_t* channel)

Stops the timer counting.

This function stops the timer from counting.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

static inline uint32_t MRT_GetIdleChannel(MRT_Type *base)

Find the available channel.

This function returns the lowest available channel number.

Parameters

- `base` – Multi-Rate timer peripheral base address

static inline void MRT_ReleaseChannel(MRT_Type *base, *mrt_chnl_t* channel)

Release the channel when the timer is using the multi-task mode.

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling

MRT_GetIdleChannel() for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number.

FSL_MRT_DRIVER_VERSION

enum *_mrt_chnl*

List of MRT channels.

Values:

enumerator kMRT_Channel_0

MRT channel number 0

enumerator kMRT_Channel_1

MRT channel number 1

enumerator kMRT_Channel_2

MRT channel number 2

enumerator kMRT_Channel_3

MRT channel number 3

enum *_mrt_timer_mode*

List of MRT timer modes.

Values:

enumerator kMRT_RepeatMode

Repeat Interrupt mode

enumerator kMRT_OneShotMode

One-shot Interrupt mode

enumerator kMRT_OneShotStallMode

One-shot stall mode

enum *_mrt_interrupt_enable*

List of MRT interrupts.

Values:

enumerator kMRT_TimerInterruptEnable

Timer interrupt enable

enum *_mrt_status_flags*

List of MRT status flags.

Values:

enumerator kMRT_TimerInterruptFlag

Timer interrupt flag

enumerator kMRT_TimerRunFlag

Indicates state of the timer

typedef enum *_mrt_chnl* *mrt_chnl_t*

List of MRT channels.

typedef enum *_mrt_timer_mode* *mrt_timer_mode_t*

List of MRT timer modes.

```
typedef enum _mrt_interrupt_enable mrt_interrupt_enable_t
```

List of MRT interrupts.

```
typedef enum _mrt_status_flags mrt_status_flags_t
```

List of MRT status flags.

```
typedef struct _mrt_config mrt_config_t
```

MRT configuration structure.

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

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

```
struct _mrt_config
```

#include <fsl_mrt.h> MRT configuration structure.

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

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

Public Members

```
bool enableMultiTask
```

true: Timers run in multi-task mode; false: Timers run in hardware status mode

2.25 PINT: Pin Interrupt and Pattern Match Driver

```
FSL_PINT_DRIVER_VERSION
```

```
enum _pint_pin_enable
```

PINT Pin Interrupt enable type.

Values:

```
enumerator kPINT_PinIntEnableNone
```

Do not generate Pin Interrupt

```
enumerator kPINT_PinIntEnableRiseEdge
```

Generate Pin Interrupt on rising edge

```
enumerator kPINT_PinIntEnableFallEdge
```

Generate Pin Interrupt on falling edge

```
enumerator kPINT_PinIntEnableBothEdges
```

Generate Pin Interrupt on both edges

```
enumerator kPINT_PinIntEnableLowLevel
```

Generate Pin Interrupt on low level

```
enumerator kPINT_PinIntEnableHighLevel
```

Generate Pin Interrupt on high level

```
enum _pint_int
```

PINT Pin Interrupt type.

Values:

enumerator kPINT_PinInt0

Pin Interrupt 0

enum _pint_pmatch_input_src

PINT Pattern Match bit slice input source type.

Values:

enumerator kPINT_PatternMatchInp0Src

Input source 0

enumerator kPINT_PatternMatchInp1Src

Input source 1

enumerator kPINT_PatternMatchInp2Src

Input source 2

enumerator kPINT_PatternMatchInp3Src

Input source 3

enumerator kPINT_PatternMatchInp4Src

Input source 4

enumerator kPINT_PatternMatchInp5Src

Input source 5

enumerator kPINT_PatternMatchInp6Src

Input source 6

enumerator kPINT_PatternMatchInp7Src

Input source 7

enumerator kPINT_SecPatternMatchInp0Src

Input source 0

enumerator kPINT_SecPatternMatchInp1Src

Input source 1

enum _pint_pmatch_bslice

PINT Pattern Match bit slice type.

Values:

enumerator kPINT_PatternMatchBSlice0

Bit slice 0

enum _pint_pmatch_bslice_cfg

PINT Pattern Match configuration type.

Values:

enumerator kPINT_PatternMatchAlways

Always Contributes to product term match

enumerator kPINT_PatternMatchStickyRise

Sticky Rising edge

enumerator kPINT_PatternMatchStickyFall

Sticky Falling edge

enumerator kPINT_PatternMatchStickyBothEdges

Sticky Rising or Falling edge

enumerator `kPINT_PatternMatchHigh`

High level

enumerator `kPINT_PatternMatchLow`

Low level

enumerator `kPINT_PatternMatchNever`

Never contributes to product term match

enumerator `kPINT_PatternMatchBothEdges`

Either rising or falling edge

typedef enum `_pint_pin_enable` `pint_pin_enable_t`

PINT Pin Interrupt enable type.

typedef enum `_pint_int` `pint_pin_int_t`

PINT Pin Interrupt type.

typedef enum `_pint_pmatch_input_src` `pint_pmatch_input_src_t`

PINT Pattern Match bit slice input source type.

typedef enum `_pint_pmatch_bslice` `pint_pmatch_bslice_t`

PINT Pattern Match bit slice type.

typedef enum `_pint_pmatch_bslice_cfg` `pint_pmatch_bslice_cfg_t`

PINT Pattern Match configuration type.

typedef struct `_pint_status` `pint_status_t`

PINT event status.

typedef void (*`pint_cb_t`)(`pint_pin_int_t` pintr, `pint_status_t` *status)

PINT Callback function.

typedef struct `_pint_pmatch_cfg` `pint_pmatch_cfg_t`

void `PINT_Init`(`PINT_Type` *base)

Initialize PINT peripheral.

This function initializes the PINT peripheral and enables the clock.

Parameters

- `base` – Base address of the PINT peripheral.

Return values

None. –

void `PINT_SetCallback`(`PINT_Type` *base, `pint_cb_t` callback)

Set PINT callback.

This function set the callback for PINT interrupt handler.

Parameters

- `base` – Base address of the PINT peripheral.
- `callback` – Callback.

Return values

None. –

void `PINT_PinInterruptConfig`(`PINT_Type` *base, `pint_pin_int_t` intr, `pint_pin_enable_t` enable)

Configure PINT peripheral pin interrupt.

This function configures a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- intr – Pin interrupt.
- enable – Selects detection logic.

Return values

None. –

```
void PINT_PinInterruptGetConfig(PINT_Type *base, pint_pin_int_t pintr, pint_pin_enable_t *enable)
```

Get PINT peripheral pin interrupt configuration.

This function returns the configuration of a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.
- enable – Pointer to store the detection logic.

Return values

None. –

```
void PINT_PinInterruptClrStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt status only when the pin was triggered by edge-sensitive.

This function clears the selected pin interrupt status.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Get Selected pin interrupt status.

This function returns the selected pin interrupt status.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

status – = 0 No pin interrupt request. = 1 Selected Pin interrupt request active.

```
void PINT_PinInterruptClrStatusAll(PINT_Type *base)
```

Clear all pin interrupts status only when pins were triggered by edge-sensitive.

This function clears the status of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatusAll(PINT_Type *base)
```

Get all pin interrupts status.

This function returns the status of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the status of corresponding pin interrupt.
= 0 No pin interrupt request. = 1 Pin interrupt request active.

```
static inline void PINT_PinInterruptClrFallFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt fall flag.

This function clears the selected pin interrupt fall flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Get selected pin interrupt fall flag.

This function returns the selected pin interrupt fall flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

flag – = 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrFallFlagAll(PINT_Type *base)
```

Clear all pin interrupt fall flags.

This function clears the fall flag for all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlagAll(PINT_Type *base)
```

Get all pin interrupt fall flags.

This function returns the fall flag of all pin interrupts.

Parameters

- base – Base address of the PINT peripheral.

Return values

flags – Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlag(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt rise flag.

This function clears the selected pin interrupt rise flag.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlag(PINT_Type *base, pintr)
```

Get selected pin interrupt rise flag.

This function returns the selected pin interrupt rise flag.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values*flag* – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlagAll(PINT_Type *base)
```

Clear all pin interrupt rise flags.

This function clears the rise flag for all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlagAll(PINT_Type *base)
```

Get all pin interrupt rise flags.

This function returns the rise flag of all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values*flags* – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
void PINT_PatternMatchConfig(PINT_Type *base, bslice, cfg)
```

Configure PINT pattern match.

This function configures a given pattern match bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.
- *cfg* – Pointer to bit slice configuration.

Return values

None. –

```
void PINT_PatternMatchGetConfig(PINT_Type *base, bslice, cfg)
```

Get PINT pattern match configuration.

This function returns the configuration of a given pattern match bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.

- *cfg* – Pointer to bit slice configuration.

Return values

None. –

```
static inline uint32_t PINT_PatternMatchGetStatus(PINT_Type *base, pint_pmatch_bslice_t bslice)
```

Get pattern match bit slice status.

This function returns the status of selected bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.

Return values

status – = 0 Match has not been detected. = 1 Match has been detected.

```
static inline uint32_t PINT_PatternMatchGetStatusAll(PINT_Type *base)
```

Get status of all pattern match bit slices.

This function returns the status of all bit slices.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the match status of corresponding bit slice.
= 0 Match has not been detected. = 1 Match has been detected.

```
uint32_t PINT_PatternMatchResetDetectLogic(PINT_Type *base)
```

Reset pattern match detection logic.

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

pmstatus – Each bit position indicates the match status of corresponding bit slice.
= 0 Match was detected. = 1 Match was not detected.

```
static inline void PINT_PatternMatchEnable(PINT_Type *base)
```

Enable pattern match function.

This function enables the pattern match function.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

```
static inline void PINT_PatternMatchDisable(PINT_Type *base)
```

Disable pattern match function.

This function disables the pattern match function.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchEnableRXEV(PINT_Type *base)

Enable RXEV output.

This function enables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchDisableRXEV(PINT_Type *base)

Disable RXEV output.

This function disables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_EnableCallback(PINT_Type *base)

Enable callback.

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_DisableCallback(PINT_Type *base)

Disable callback.

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

- base – Base address of the peripheral.

Return values

None. –

void PINT_Deinit(PINT_Type *base)

Deinitialize PINT peripheral.

This function disables the PINT clock.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_EnableCallbackByIndex(PINT_Type *base, *pint_pin_int_t* pintIdx)

enable callback by pin index.

This function enables callback by pin index instead of enabling all pins.

Parameters

- base – Base address of the peripheral.

- pintIdx – pin index.

Return values

None. –

`void PINT_DisableCallbackByIndex(PINT_Type *base, pint_pin_int_t pintIdx)`
disable callback by pin index.

This function disables callback by pin index instead of disabling all pins.

Parameters

- base – Base address of the peripheral.
- pintIdx – pin index.

Return values

None. –

`PINT_USE_LEGACY_CALLBACK`

`PININT_BITSLICE_SRC_START`

`PININT_BITSLICE_SRC_MASK`

`PININT_BITSLICE_CFG_START`

`PININT_BITSLICE_CFG_MASK`

`PININT_BITSLICE_ENDP_MASK`

`PINT_PIN_INT_LEVEL`

`PINT_PIN_INT_EDGE`

`PINT_PIN_INT_FALL_OR_HIGH_LEVEL`

`PINT_PIN_INT_RISE`

`PINT_PIN_RISE_EDGE`

`PINT_PIN_FALL_EDGE`

`PINT_PIN_BOTH_EDGE`

`PINT_PIN_LOW_LEVEL`

`PINT_PIN_HIGH_LEVEL`

`struct __pint_status`

#include <fsl_pint.h> PINT event status.

`struct __pint_pmatch_cfg`

#include <fsl_pint.h>

2.26 Power Driver

`enum pd_bits`

Values:

enumerator `kPDRUNCFG_PD_FRO_EN`

enumerator `kPDRUNCFG_PD_FLASH`

enumerator kPDRUNCFG_PD_TEMPS
enumerator kPDRUNCFG_PD_BOD_RESET
enumerator kPDRUNCFG_PD_BOD_INTR
enumerator kPDRUNCFG_PD_ADC0
enumerator kPDRUNCFG_PD_VDDFLASH
enumerator kPDRUNCFG_LP_VDDFLASH
enumerator kPDRUNCFG_PD_RAM0
enumerator kPDRUNCFG_PD_RAM1
enumerator kPDRUNCFG_PD_RAM2
enumerator kPDRUNCFG_PD_RAMX
enumerator kPDRUNCFG_PD_ROM
enumerator kPDRUNCFG_PD_VDDHV_ENA
enumerator kPDRUNCFG_PD_VD7_ENA
enumerator kPDRUNCFG_PD_WDT_OSC
enumerator kPDRUNCFG_PD_USB0_PHY
enumerator kPDRUNCFG_PD_SYS_PLL0
enumerator kPDRUNCFG_PD_VREFP_SW
enumerator kPDRUNCFG_PD_FLASH_BG
enumerator kPDRUNCFG_PD_ALT_FLASH_IBG
enumerator kPDRUNCFG_SEL_ALT_FLASH_IBG
enumerator kPDRUNCFG_ForceUnsigned

enum __power_mode_config

Values:

enumerator kPmu_Sleep
enumerator kPmu_Deep_Sleep
enumerator kPmu_Deep_PowerDown

enum __power_bod_status

The enumeration of BOD status flags.

Values:

enumerator kBod_ResetStatusFlag
BOD reset has occurred.
enumerator kBod_InterruptStatusFlag
BOD interrupt has occurred

enum __power_bod_reset_level

The enumeration of BOD reset level.

Values:

enumerator kBod_ResetLevel0

Reset Level0: 1.5V.

enumerator kBod_ResetLevel1

Reset Level0: 1.85V.

enumerator kBod_ResetLevel2

Reset Level0: 2.0V.

enumerator kBod_ResetLevel3

Reset Level0: 2.3V.

enum __power_bod_interrupt_level

The enumeration of BOD interrupt level.

Values:

enumerator kBod_InterruptLevel0

Interrupt level: 2.05V.

enumerator kBod_InterruptLevel1

Interrupt level: 2.45V.

enumerator kBod_InterruptLevel2

Interrupt level: 2.75V.

enumerator kBod_InterruptLevel3

Interrupt level: 3.05V.

typedef enum *pd_bits* pd_bit_t

typedef enum *__power_mode_config* power_mode_cfg_t

typedef enum *__power_bod_status* power_bod_status_t

The enumeration of BOD status flags.

typedef enum *__power_bod_reset_level* power_bod_reset_level_t

The enumeration of BOD reset level.

typedef enum *__power_bod_interrupt_level* power_bod_interrupt_level_t

The enumeration of BOD interrupt level.

typedef struct *__power_bod_config* power_bod_config_t

The configuration of power bod, including reset level, interrupt level, and so on.

FSL_POWER_DRIVER_VERSION

power driver version 2.1.0.

MAKE_PD_BITS(reg, slot)

PDRCFG0

PDRCFG1

static inline void POWER_EnablePD(*pd_bit_t* en)

API to enable PDRUNCFG bit in the Syscon. Note that enabling the bit powers down the peripheral.

Parameters

- en – peripheral for which to enable the PDRUNCFG bit

Returns

none

static inline void POWER_DisablePD(*pd_bit_t* en)

API to disable PDRUNCFG bit in the Syscon. Note that disabling the bit powers up the peripheral.

Parameters

- en – peripheral for which to disable the PDRUNCFG bit

Returns

none

static inline void POWER_EnableDeepSleep(void)

API to enable deep sleep bit in the ARM Core.

Returns

none

static inline void POWER_DisableDeepSleep(void)

API to disable deep sleep bit in the ARM Core.

Returns

none

static inline void POWER_PowerDownFlash(void)

API to power down flash controller.

Returns

none

static inline void POWER_PowerUpFlash(void)

API to power up flash controller.

Returns

none

void POWER_EnterPowerMode(*power_mode_cfg_t* mode, uint64_t exclude_from_pd)

Power Library API to enter different power mode.

Parameters

- mode –
- exclude_from_pd – Bit mask of the PDRUNCFG bits that needs to be powered on during deep sleep

Returns

none

void POWER_EnterSleep(void)

Power Library API to enter sleep mode.

Returns

none

void POWER_EnterDeepSleep(uint64_t exclude_from_pd)

Power Library API to enter deep sleep mode.

Parameters

- exclude_from_pd – Bit mask of the PDRUNCFG bits that needs to be powered on during deep sleep

Returns

none

`void POWER_EnterDeepPowerDown(uint64_t exclude_from_pd)`

Power Library API to enter deep power down mode.

Parameters

- `exclude_from_pd` – Bit mask of the PDRUNCFG bits that needs to be powered on during deep power down mode, but this has no effect as the voltages are cut off.

Returns

none

`void POWER_SetVoltageForFreq(uint32_t freq)`

Power Library API to choose normal regulation and set the voltage for the desired operating frequency.

Parameters

- `freq` – The desired frequency at which the part would like to operate, note that the voltage and flash wait states should be set before changing frequency

Returns

none

`void POWER_SetLowPowerVoltageForFreq(uint32_t freq)`

Power Library API to choose low power regulation and set the voltage for the desired operating frequency.

Parameters

- `freq` – The desired frequency at which the part would like to operate, note only 12MHz and 48Mhz are supported

Returns

none

`uint32_t POWER_GetLibVersion(void)`

Power Library API to return the library version.

Returns

version number of the power library

`void POWER_InitBod(const power_bod_config_t *bodConfig)`

Initialize BOD, including enabling/disabling BOD interrupt, enabling/disabling BOD reset, setting BOD interrupt level, and reset level.

Parameters

- `bodConfig` – Pointer to the structure `power_bod_config_t`.

`void POWER_GetDefaultBodConfig(power_bod_config_t *bodConfig)`

Get default BOD configuration.

```
bodConfig->enableReset = true;
bodConfig->resetLevel = kBod_ResetLevel0;
bodConfig->enableInterrupt = false;
bodConfig->interruptLevel = kBod_InterruptLevel0;
```

Parameters

- `bodConfig` – Pointer to the structure `power_bod_config_t`.

`static inline void POWER_DeinitBod(void)`

De-initialize BOD.

```
static inline uint32_t POWER_GetBodStatusFlags(void)
```

Get Bod status flags.

Returns

Flags of Bod status.

```
static inline void POWER_ClearBodStatusFlags(uint32_t mask)
```

Clear Bod status flags.

Parameters

- `mask` – The mask of status flags to clear, should be the OR'ed value of `power_bod_status_t`.

```
bool enableReset
```

Enable/disable BOD reset function.

```
power_bod_reset_level_t resetLevel
```

BOD reset level, please refer to `power_bod_reset_level_t`.

```
bool enableInterrupt
```

Enable/disable BOD interrupt function.

```
power_bod_interrupt_level_t interruptLevel
```

BOD interrupt level, please refer to `power_bod_interrupt_level_t`.

```
struct __power_bod_config
```

`#include <fsl_power.h>` The configuration of power bod, including reset level, interrupt level, and so on.

2.27 Reset Driver

```
enum _SYSCON_RSTn
```

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

Values:

```
enumerator kRSTn_IpInvalid
```

```
enumerator kFLASH_RST_SHIFT_RSTn
```

Flash controller reset control

```
enumerator kFMC_RST_SHIFT_RSTn
```

Flash accelerator reset control

```
enumerator kMUX_RST_SHIFT_RSTn
```

Input mux reset control

```
enumerator kIOCON_RST_SHIFT_RSTn
```

IOCON reset control

```
enumerator kGPIO0_RST_SHIFT_RSTn
```

GPIO0 reset control

```
enumerator kGPIO1_RST_SHIFT_RSTn
```

GPIO1 reset control

enumerator kPINT_RST_SHIFT_RSTn
Pin interrupt (PINT) reset control

enumerator kGINT_RST_SHIFT_RSTn
Grouped interrupt (PINT) reset control.

enumerator kDMA_RST_SHIFT_RSTn
DMA reset control

enumerator kCRC_RST_SHIFT_RSTn
CRC reset control

enumerator kWWDT_RST_SHIFT_RSTn
Watchdog timer reset control

enumerator kADC0_RST_SHIFT_RSTn
ADC0 reset control

enumerator kMRT_RST_SHIFT_RSTn
Multi-rate timer (MRT) reset control

enumerator kSCT0_RST_SHIFT_RSTn
SCTimer/PWM 0 (SCT0) reset control

enumerator kUTICK_RST_SHIFT_RSTn
Micro-tick timer reset control

enumerator kFC0_RST_SHIFT_RSTn
Flexcomm Interface 0 reset control

enumerator kFC1_RST_SHIFT_RSTn
Flexcomm Interface 1 reset control

enumerator kFC2_RST_SHIFT_RSTn
Flexcomm Interface 2 reset control

enumerator kFC3_RST_SHIFT_RSTn
Flexcomm Interface 3 reset control

enumerator kFC4_RST_SHIFT_RSTn
Flexcomm Interface 4 reset control

enumerator kFC5_RST_SHIFT_RSTn
Flexcomm Interface 5 reset control

enumerator kFC6_RST_SHIFT_RSTn
Flexcomm Interface 6 reset control

enumerator kFC7_RST_SHIFT_RSTn
Flexcomm Interface 7 reset control

enumerator kUSB_RST_SHIFT_RSTn
USB reset control

enumerator kCTIMER0_RST_SHIFT_RSTn
CTimer0 reset control

enumerator kCTIMER1_RST_SHIFT_RSTn
CTimer1 reset control

enumerator kCTIMER3_RST_SHIFT_RSTn
CTimer3 reset control

```
typedef enum _SYSCON_RSTn SYSCON_RSTn_t
```

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

```
typedef SYSCON_RSTn_t reset_ip_name_t
```

```
void RESET_SetPeripheralReset(reset_ip_name_t peripheral)
```

Assert reset to peripheral.

Asserts reset signal to specified peripheral module.

Parameters

- peripheral – Assert reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

```
void RESET_ClearPeripheralReset(reset_ip_name_t peripheral)
```

Clear reset to peripheral.

Clears reset signal to specified peripheral module, allows it to operate.

Parameters

- peripheral – Clear reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

```
void RESET_PeripheralReset(reset_ip_name_t peripheral)
```

Reset peripheral module.

Reset peripheral module.

Parameters

- peripheral – Peripheral to reset. The enum argument contains encoding of reset register and reset bit position in the reset register.

```
static inline void RESET_ReleasePeripheralReset(reset_ip_name_t peripheral)
```

Release peripheral module.

Release peripheral module.

Parameters

- peripheral – Peripheral to release. The enum argument contains encoding of reset register and reset bit position in the reset register.

```
FSL_RESET_DRIVER_VERSION
```

reset driver version 2.4.0

```
ADC_RSTS
```

Array initializers with peripheral reset bits

```
CRC_RSTS
```

```
DMA_RSTS_N
```

```
FLEXCOMM_RSTS
```

```
GINT_RSTS
```

```
GPIO_RSTS_N
```

```
INPUTMUX_RSTS
```

```
IOCON_RSTS
```

FLASH_RSTS

MRT_RSTS

PINT_RSTS

SCT_RSTS

CTIMER_RSTS

USB_RSTS

UTICK_RSTS

WWDT_RSTS

2.28 RTC: Real Time Clock

`void RTC_Init(RTC_Type *base)`

Un-gate the RTC clock and enable the RTC oscillator.

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

Parameters

- `base` – RTC peripheral base address

`static inline void RTC_Deinit(RTC_Type *base)`

Stop the timer and gate the RTC clock.

Parameters

- `base` – RTC peripheral base address

`status_t RTC_SetDatetime(RTC_Type *base, const rtc_datetime_t *datetime)`

Set the RTC date and time according to the given time structure.

The RTC counter must be stopped prior to calling this function as writes to the RTC seconds register will fail if the RTC counter is running.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to structure where the date and time details to set are stored

Returns

`kStatus_Success`: Success in setting the time and starting the RTC
`kStatus_InvalidArgument`: Error because the datetime format is incorrect

`void RTC_GetDatetime(RTC_Type *base, rtc_datetime_t *datetime)`

Get the RTC time and stores it in the given time structure.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to structure where the date and time details are stored.

```
status_t RTC_SetAlarm(RTC_Type *base, const rtc_datetime_t *alarmTime)
```

Set the RTC alarm time.

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

- `base` – RTC peripheral base address
- `alarmTime` – Pointer to structure where the alarm time is stored.

Returns

`kStatus_Success`: success in setting the RTC alarm
`kStatus_InvalidArgument`: Error because the alarm datetime format is incorrect
`kStatus_Fail`: Error because the alarm time has already passed

```
void RTC_GetAlarm(RTC_Type *base, rtc_datetime_t *datetime)
```

Return the RTC alarm time.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to structure where the alarm date and time details are stored.

```
static inline void RTC_EnableWakeupTimer(RTC_Type *base, bool enable)
```

Enable the RTC wake-up timer (1KHZ).

After calling this function, the RTC driver will use/un-use the RTC wake-up (1KHZ) at the same time.

Parameters

- `base` – RTC peripheral base address
- `enable` – Use/Un-use the RTC wake-up timer.
 - `true`: Use RTC wake-up timer at the same time.
 - `false`: Un-use RTC wake-up timer, RTC only use the normal seconds timer by default.

```
static inline uint32_t RTC_GetEnabledWakeupTimer(RTC_Type *base)
```

Get the enabled status of the RTC wake-up timer (1KHZ).

Parameters

- `base` – RTC peripheral base address

Returns

The enabled status of RTC wake-up timer (1KHZ).

```
static inline void RTC_EnableWakeUpTimerInterruptFromDPD(RTC_Type *base, bool enable)
```

Enable the wake-up timer interrupt from deep power down mode.

Parameters

- `base` – RTC peripheral base address
- `enable` – Enable/Disable wake-up timer interrupt from deep power down mode.
 - `true`: Enable wake-up timer interrupt from deep power down mode.
 - `false`: Disable wake-up timer interrupt from deep power down mode.

static inline void RTC_EnableAlarmTimerInterruptFromDPD(RTC_Type *base, bool enable)

Enable the alarm timer interrupt from deep power down mode.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable alarm timer interrupt from deep power down mode.
 - true: Enable alarm timer interrupt from deep power down mode.
 - false: Disable alarm timer interrupt from deep power down mode.

static inline void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superceded by RTC_EnableAlarmTimerInterruptFromDPD and RTC_EnableWakeUpTimerInterruptFromDPD

Parameters

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

static inline void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superceded by RTC_EnableAlarmTimerInterruptFromDPD and RTC_EnableWakeUpTimerInterruptFromDPD

Parameters

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

static inline uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)

Get the enabled RTC interrupts.

Deprecated:

Do not use this function. It will be deleted in next release version.

Parameters

- base – RTC peripheral base address

Returns

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

static inline uint32_t RTC_GetStatusFlags(RTC_Type *base)

Get the RTC status flags.

Parameters

- base – RTC peripheral base address

Returns

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

```
static inline void RTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)
```

Clear the RTC status flags.

Parameters

- `base` – RTC peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `rtc_status_flags_t`

```
static inline void RTC_EnableTimer(RTC_Type *base, bool enable)
```

Enable the RTC timer counter.

After calling this function, the RTC inner counter increments once a second when only using the RTC seconds timer (1hz), while the RTC inner wake-up timer countdown once a millisecond when using RTC wake-up timer (1KHZ) at the same time. RTC timer contain two timers, one is the RTC normal seconds timer, the other one is the RTC wake-up timer, the RTC enable bit is the master switch for the whole RTC timer, so user can use the RTC seconds (1HZ) timer independently, but they can't use the RTC wake-up timer (1KHZ) independently.

Parameters

- `base` – RTC peripheral base address
- `enable` – Enable/Disable RTC Timer counter.
 - `true`: Enable RTC Timer counter.
 - `false`: Disable RTC Timer counter.

```
static inline void RTC_StartTimer(RTC_Type *base)
```

Starts the RTC time counter.

Deprecated:

Do not use this function. It has been superceded by `RTC_EnableTimer`

After calling this function, the timer counter increments once a second provided `SR[TOF]` or `SR[TIF]` are not set.

Parameters

- `base` – RTC peripheral base address

```
static inline void RTC_StopTimer(RTC_Type *base)
```

Stops the RTC time counter.

Deprecated:

Do not use this function. It has been superceded by `RTC_EnableTimer`

RTC's seconds register can be written to only when the timer is stopped.

Parameters

- `base` – RTC peripheral base address

FSL_RTC_DRIVER_VERSION

Version 2.2.0

enum `_rtc_interrupt_enable`

List of RTC interrupts.

Values:

enumerator `kRTC_AlarmInterruptEnable`

Alarm interrupt.

enumerator `kRTC_WakeupInterruptEnable`

Wake-up interrupt.

enum `_rtc_status_flags`

List of RTC flags.

Values:

enumerator `kRTC_AlarmFlag`

Alarm flag

enumerator `kRTC_WakeupFlag`

1kHz wake-up timer flag

typedef enum `_rtc_interrupt_enable` `rtc_interrupt_enable_t`

List of RTC interrupts.

typedef enum `_rtc_status_flags` `rtc_status_flags_t`

List of RTC flags.

typedef struct `_rtc_datetime` `rtc_datetime_t`

Structure is used to hold the date and time.

static inline void `RTC_SetSecondsTimerMatch(RTC_Type *base, uint32_t matchValue)`

Set the RTC seconds timer (1HZ) MATCH value.

Parameters

- `base` – RTC peripheral base address
- `matchValue` – The value to be set into the RTC MATCH register

static inline uint32_t `RTC_GetSecondsTimerMatch(RTC_Type *base)`

Read actual RTC seconds timer (1HZ) MATCH value.

Parameters

- `base` – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) MATCH value.

static inline void `RTC_SetSecondsTimerCount(RTC_Type *base, uint32_t countValue)`

Set the RTC seconds timer (1HZ) COUNT value.

Parameters

- `base` – RTC peripheral base address
- `countValue` – The value to be loaded into the RTC COUNT register

static inline uint32_t `RTC_GetSecondsTimerCount(RTC_Type *base)`

Read the actual RTC seconds timer (1HZ) COUNT value.

Parameters

- `base` – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) COUNT value.

```
static inline void RTC_SetWakeupCount(RTC_Type *base, uint16_t wakeupValue)
```

Enable the RTC wake-up timer (1KHZ) and set countdown value to the RTC WAKE register.

Parameters

- base – RTC peripheral base address
- wakeupValue – The value to be loaded into the WAKE register in RTC wake-up timer (1KHZ).

```
static inline uint16_t RTC_GetWakeupCount(RTC_Type *base)
```

Read the actual value from the WAKE register value in RTC wake-up timer (1KHZ)

Read the WAKE register twice and compare the result, if the value match, the time can be used.

Parameters

- base – RTC peripheral base address

Returns

The actual value of the WAKE register value in RTC wake-up timer (1KHZ).

```
static inline void RTC_Reset(RTC_Type *base)
```

Perform a software reset on the RTC module.

This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.

Parameters

- base – RTC peripheral base address

```
struct _rtc_datetime
```

#include <fsl_rtc.h> Structure is used to hold the date and time.

Public Members

uint16_t year

Range from 1970 to 2099.

uint8_t month

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

2.29 SCTimer: SCTimer/PWM (SCT)

status_t SCTIMER_Init(SCT_Type *base, const *sctimer_config_t* *config)

Ungates the SCTimer clock and configures the peripheral for basic operation.

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

Parameters

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

Returns

kStatus_Success indicates success; Else indicates failure.

void SCTIMER_Deinit(SCT_Type *base)

Gates the SCTimer clock.

Parameters

- base – SCTimer peripheral base address

void SCTIMER_GetDefaultConfig(*sctimer_config_t* *config)

Fills in the SCTimer configuration structure with the default settings.

The default values are:

```
config->enableCounterUnify = true;
config->clockMode = kSCTIMER_System_ClockMode;
config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
config->enableBidirection_l = false;
config->enableBidirection_h = false;
config->prescale_l = 0U;
config->prescale_h = 0U;
config->outInitState = 0U;
config->inputsync = 0xFU;
```

Parameters

- config – Pointer to the user configuration structure.

status_t SCTIMER_SetupPwm(SCT_Type *base, const *sctimer_pwm_signal_param_t* *pwmParams, *sctimer_pwm_mode_t* mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, uint32_t *event)

Configures the PWM signal parameters.

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function SCTIMER_GetCurrentStateNumber(). The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

Note: When setting PWM output from multiple output pins, they all should use the same PWM mode i.e all PWM's should be either edge-aligned or center-aligned. When using this API, the PWM signal frequency of all the initialized channels must be the same. Otherwise all the initialized channels' PWM signal frequency is equal to the last call to the API's pwmFreq_Hz.

Parameters

- `base` – SCTimer peripheral base address
- `pwmParams` – PWM parameters to configure the output
- `mode` – PWM operation mode, options available in enumeration `sctimer_pwm_mode_t`
- `pwmFreq_Hz` – PWM signal frequency in Hz
- `srcClock_Hz` – SCTimer counter clock in Hz
- `event` – Pointer to a variable where the PWM period event number is stored

Returns

`kStatus_Success` on success `kStatus_Fail` If we have hit the limit in terms of number of events created or if an incorrect PWM duty cycle is passed in.

```
void SCTIMER_UpdatePwmDutycycle(SCT_Type *base, sctimer_out_t output, uint8_t  
                                dutyCyclePercent, uint32_t event)
```

Updates the duty cycle of an active PWM signal.

Before calling this function, the counter is set to operate as one 32-bit counter (unify bit is set to 1).

Parameters

- `base` – SCTimer peripheral base address
- `output` – The output to configure
- `dutyCyclePercent` – New PWM pulse width; the value should be between 1 to 100
- `event` – Event number associated with this PWM signal. This was returned to the user by the function `SCTIMER_SetupPwm()`.

```
static inline void SCTIMER_EnableInterrupts(SCT_Type *base, uint32_t mask)
```

Enables the selected SCTimer interrupts.

Parameters

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

```
static inline void SCTIMER_DisableInterrupts(SCT_Type *base, uint32_t mask)
```

Disables the selected SCTimer interrupts.

Parameters

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

```
static inline uint32_t SCTIMER_GetEnabledInterrupts(SCT_Type *base)
```

Gets the enabled SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address

Returns

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

```
static inline uint32_t SCTIMER_GetStatusFlags(SCT_Type *base)
```

Gets the SCTimer status flags.

Parameters

- base – SCTimer peripheral base address

Returns

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

```
static inline void SCTIMER_ClearStatusFlags(SCT_Type *base, uint32_t mask)
```

Clears the SCTimer status flags.

Parameters

- base – SCTimer peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `sctimer_status_flags_t`

```
static inline void SCTIMER_StartTimer(SCT_Type *base, uint32_t countertoStart)
```

Starts the SCTimer counter.

Note: In 16-bit mode, we can enable both Counter_L and Counter_H, In 32-bit mode, we only can select Counter_U.

Parameters

- base – SCTimer peripheral base address
- countertoStart – The SCTimer counters to enable. This is a logical OR of members of the enumeration `sctimer_counter_t`.

```
static inline void SCTIMER_StopTimer(SCT_Type *base, uint32_t countertoStop)
```

Halts the SCTimer counter.

Parameters

- base – SCTimer peripheral base address
- countertoStop – The SCTimer counters to stop. This is a logical OR of members of the enumeration `sctimer_counter_t`.

```
status_t SCTIMER_CreateAndScheduleEvent(SCT_Type *base, sctimer_event_t howToMonitor,  
                                         uint32_t matchValue, uint32_t whichIO,  
                                         sctimer_counter_t whichCounter, uint32_t *event)
```

Create an event that is triggered on a match or IO and schedule in current state.

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

Parameters

- base – SCTimer peripheral base address
- howToMonitor – Event type; options are available in the enumeration `sctimer_interrupt_enable_t`
- matchValue – The match value that will be programmed to a match register

- `whichIO` – The input or output that will be involved in event triggering. This field is ignored if the event type is “match only”
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Pointer to a variable where the new event number is stored

Returns

`kStatus_Success` on success `kStatus_Error` if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

```
void SCTIMER_ScheduleEvent(SCT_Type *base, uint32_t event)
```

Enable an event in the current state.

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function `SCTIMER_SetupPwm()` or function `SCTIMER_CreateAndScheduleEvent()`.

Parameters

- `base` – SCTimer peripheral base address
- `event` – Event number to enable in the current state

```
status_t SCTIMER_IncreaseState(SCT_Type *base)
```

Increase the state by 1.

All future events created by calling the function `SCTIMER_ScheduleEvent()` will be enabled in this new state.

Parameters

- `base` – SCTimer peripheral base address

Returns

`kStatus_Success` on success `kStatus_Error` if we have hit the limit in terms of states used

```
uint32_t SCTIMER_GetCurrentState(SCT_Type *base)
```

Provides the current state.

User can use this to set the next state by calling the function `SCTIMER_SetupNextStateAction()`.

Parameters

- `base` – SCTimer peripheral base address

Returns

The current state

```
static inline void SCTIMER_SetCounterState(SCT_Type *base, sctimer_counter_t whichCounter,
                                           uint32_t state)
```

Set the counter current state.

The function is to set the state variable bit field of STATE register. Writing to the `STATE_L`, `STATE_H`, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `state` – The counter current state number (only support range from 0~31).

```
static inline uint16_t SCTIMER_GetCounterState(SCT_Type *base, sctimer_counter_t
                                              whichCounter)
```

Get the counter current state value.

The function is to get the state variable bit field of STATE register.

Parameters

- *base* – SCTimer peripheral base address
- *whichCounter* – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The the counter current state value.

```
status_t SCTIMER_SetupCaptureAction(SCT_Type *base, sctimer_counter_t whichCounter,
                                   uint32_t *captureRegister, uint32_t event)
```

Setup capture of the counter value on trigger of a selected event.

Parameters

- *base* – SCTimer peripheral base address
- *whichCounter* – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- *captureRegister* – Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
- *event* – Event number that will trigger the capture

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

```
void SCTIMER_SetCallback(SCT_Type *base, sctimer_event_callback_t callback, uint32_t event)
```

Receive notification when the event trigger an interrupt.

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

Parameters

- *base* – SCTimer peripheral base address
- *event* – Event number that will trigger the interrupt
- *callback* – Function to invoke when the event is triggered

```
static inline void SCTIMER_SetupStateLdMethodAction(SCT_Type *base, uint32_t event, bool
                                                    fgLoad)
```

Change the load method of transition to the specified state.

Change the load method of transition, it will be triggered by the event number that is passed in by the user.

Parameters

- *base* – SCTimer peripheral base address
- *event* – Event number that will change the method to trigger the state transition
- *fgLoad* – The method to load highest-numbered event occurring for that state to the STATE register.

- true: Load the STATEV value to STATE when the event occurs to be the next state.
- false: Add the STATEV value to STATE when the event occurs to be the next state.

```
static inline void SCTIMER_SetupNextStateActionwithLdMethod(SCT_Type *base, uint32_t
                                                         nextState, uint32_t event, bool
                                                         fgLoad)
```

Transition to the specified state with Load method.

This transition will be triggered by the event number that is passed in by the user; the method decide how to load the highest-numbered event occurring for that state to the STATE register.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition
- fgLoad – The method to load the highest-numbered event occurring for that state to the STATE register.
 - true: Load the STATEV value to STATE when the event occurs to be the next state.
 - false: Add the STATEV value to STATE when the event occurs to be the next state.

```
static inline void SCTIMER_SetupNextStateAction(SCT_Type *base, uint32_t nextState, uint32_t
                                              event)
```

Transition to the specified state.

Deprecated:

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

This transition will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition

```
static inline void SCTIMER_SetupEventActiveDirection(SCT_Type *base,
                                                    sctimer_event_active_direction_t
                                                    activeDirection, uint32_t event)
```

Setup event active direction when the counters are operating in BIDIR mode.

Parameters

- base – SCTimer peripheral base address
- activeDirection – Event generation active direction, see sctimer_event_active_direction_t.
- event – Event number that need setup the active direction.

```
static inline void SCTIMER_SetupOutputSetAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Set the Output.

This output will be set when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to set
- event – Event number that will trigger the output change

```
static inline void SCTIMER_SetupOutputClearAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Clear the Output.

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to clear
- event – Event number that will trigger the output change

```
void SCTIMER_SetupOutputToggleAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Toggle the output level.

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to toggle
- event – Event number that will trigger the output change

```
static inline void SCTIMER_SetupCounterLimitAction(SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)
```

Limit the running counter.

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be limited

```
static inline void SCTIMER_SetupCounterStopAction(SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)
```

Stop the running counter.

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be stopped

```
static inline void SCTIMER_SetupCounterStartAction(SCT_Type *base, sctimer_counter_t
                                                    whichCounter, uint32_t event)
```

Re-start the stopped counter.

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to re-start

```
static inline void SCTIMER_SetupCounterHaltAction(SCT_Type *base, sctimer_counter_t
                                                    whichCounter, uint32_t event)
```

Halt the running counter.

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the SCTIMER_StartTimer() function.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be halted

```
static inline void SCTIMER_SetupDmaTriggerAction(SCT_Type *base, uint32_t dmaNumber,
                                                  uint32_t event)
```

Generate a DMA request.

DMA request will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- dmaNumber – The DMA request to generate
- event – Event number that will trigger the DMA request

```
static inline void SCTIMER_SetCOUNTValue(SCT_Type *base, sctimer_counter_t whichCounter,
                                           uint32_t value)
```

Set the value of counter.

The function is to set the value of Count register, Writing to the COUNT_L, COUNT_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- value – the counter value update to the COUNT register.

```
static inline uint32_t SCTIMER_GetCOUNTValue(SCT_Type *base, sctimer_counter_t
                                                whichCounter)
```

Get the value of counter.

The function is to read the value of Count register, software can read the counter registers at any time..

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The value of counter selected.

```
static inline void SCTIMER_SetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Set the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be set.
- state – The state value in which the event is enabled to occur.

```
static inline void SCTIMER_ClearEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Clear the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be clear.
- state – The state value in which the event is disabled to occur.

```
static inline bool SCTIMER_GetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Get the state mask bit field of EV_STATE register.

Note: This function is to check whether the event is enabled in a specific state.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be read.
- state – The state value.

Returns

The the state mask bit field of EV_STATE register.

- true: The event is enable in state.
- false: The event is disable in state.

```
static inline uint32_t SCTIMER_GetCaptureValue(SCT_Type *base, sctimer_counter_t  
                                              whichCounter, uint8_t capChannel)
```

Get the value of capture register.

This function returns the captured value upon occurrence of the events selected by the corresponding Capture Control registers occurred.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- capChannel – SCTimer capture register of capture channel.

Returns

The SCTimer counter value at which this register was last captured.

void SCTIMER_EventHandleIRQ(SCT_Type *base)

SCTimer interrupt handler.

Parameters

- base – SCTimer peripheral base address.

FSL_SCTIMER_DRIVER_VERSION

Version

enum _sctimer_pwm_mode

SCTimer PWM operation modes.

Values:

enumerator kSCTIMER_EdgeAlignedPwm

Edge-aligned PWM

enumerator kSCTIMER_CenterAlignedPwm

Center-aligned PWM

enum _sctimer_counter

SCTimer counters type.

Values:

enumerator kSCTIMER_Counter_L

16-bit Low counter.

enumerator kSCTIMER_Counter_H

16-bit High counter.

enumerator kSCTIMER_Counter_U

32-bit Unified counter.

enum _sctimer_input

List of SCTimer input pins.

Values:

enumerator kSCTIMER_Input_0

SCTIMER input 0

enumerator kSCTIMER_Input_1

SCTIMER input 1

enumerator kSCTIMER_Input_2

SCTIMER input 2

enumerator kSCTIMER_Input_3

SCTIMER input 3

enumerator kSCTIMER_Input_4

SCTIMER input 4

enumerator kSCTIMER_Input_5

SCTIMER input 5

enumerator kSCTIMER_Input_6

SCTIMER input 6

enumerator kSCTIMER_Input_7
SCTIMER input 7

enum _sctimer_out

List of SCTimer output pins.

Values:

enumerator kSCTIMER_Out_0
SCTIMER output 0

enumerator kSCTIMER_Out_1
SCTIMER output 1

enumerator kSCTIMER_Out_2
SCTIMER output 2

enumerator kSCTIMER_Out_3
SCTIMER output 3

enumerator kSCTIMER_Out_4
SCTIMER output 4

enumerator kSCTIMER_Out_5
SCTIMER output 5

enumerator kSCTIMER_Out_6
SCTIMER output 6

enumerator kSCTIMER_Out_7
SCTIMER output 7

enumerator kSCTIMER_Out_8
SCTIMER output 8

enumerator kSCTIMER_Out_9
SCTIMER output 9

enum _sctimer_pwm_level_select

SCTimer PWM output pulse mode: high-true, low-true or no output.

Values:

enumerator kSCTIMER_LowTrue
Low true pulses

enumerator kSCTIMER_HighTrue
High true pulses

enum _sctimer_clock_mode

SCTimer clock mode options.

Values:

enumerator kSCTIMER_System_ClockMode
System Clock Mode

enumerator kSCTIMER_Sampled_ClockMode
Sampled System Clock Mode

enumerator kSCTIMER_Input_ClockMode
SCT Input Clock Mode

enumerator kSCTIMER_Asynchronous_ClockMode
Asynchronous Mode

enum __sctimer_clock_select
SCTimer clock select options.

Values:

enumerator kSCTIMER_Clock_On_Rise_Input_0
Rising edges on input 0

enumerator kSCTIMER_Clock_On_Fall_Input_0
Falling edges on input 0

enumerator kSCTIMER_Clock_On_Rise_Input_1
Rising edges on input 1

enumerator kSCTIMER_Clock_On_Fall_Input_1
Falling edges on input 1

enumerator kSCTIMER_Clock_On_Rise_Input_2
Rising edges on input 2

enumerator kSCTIMER_Clock_On_Fall_Input_2
Falling edges on input 2

enumerator kSCTIMER_Clock_On_Rise_Input_3
Rising edges on input 3

enumerator kSCTIMER_Clock_On_Fall_Input_3
Falling edges on input 3

enumerator kSCTIMER_Clock_On_Rise_Input_4
Rising edges on input 4

enumerator kSCTIMER_Clock_On_Fall_Input_4
Falling edges on input 4

enumerator kSCTIMER_Clock_On_Rise_Input_5
Rising edges on input 5

enumerator kSCTIMER_Clock_On_Fall_Input_5
Falling edges on input 5

enumerator kSCTIMER_Clock_On_Rise_Input_6
Rising edges on input 6

enumerator kSCTIMER_Clock_On_Fall_Input_6
Falling edges on input 6

enumerator kSCTIMER_Clock_On_Rise_Input_7
Rising edges on input 7

enumerator kSCTIMER_Clock_On_Fall_Input_7
Falling edges on input 7

enum __sctimer_conflict_resolution
SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Values:

enumerator kSCTIMER_ResolveNone

No change

enumerator kSCTIMER_ResolveSet

Set output

enumerator kSCTIMER_ResolveClear

Clear output

enumerator kSCTIMER_ResolveToggle

Toggle output

enum _sctimer_event_active_direction

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

Values:

enumerator kSCTIMER_ActiveIndependent

This event is triggered regardless of the count direction.

enumerator kSCTIMER_ActiveInCountUp

This event is triggered only during up-counting when BIDIR = 1.

enumerator kSCTIMER_ActiveInCountDown

This event is triggered only during down-counting when BIDIR = 1.

enum _sctimer_event

List of SCTimer event types.

Values:

enumerator kSCTIMER_InputLowOrMatchEvent

enumerator kSCTIMER_InputRiseOrMatchEvent

enumerator kSCTIMER_InputFallOrMatchEvent

enumerator kSCTIMER_InputHighOrMatchEvent

enumerator kSCTIMER_MatchEventOnly

enumerator kSCTIMER_InputLowEvent

enumerator kSCTIMER_InputRiseEvent

enumerator kSCTIMER_InputFallEvent

enumerator kSCTIMER_InputHighEvent

enumerator kSCTIMER_InputLowAndMatchEvent

enumerator kSCTIMER_InputRiseAndMatchEvent

enumerator kSCTIMER_InputFallAndMatchEvent

enumerator kSCTIMER_InputHighAndMatchEvent

enumerator kSCTIMER_OutputLowOrMatchEvent

enumerator kSCTIMER_OutputRiseOrMatchEvent

enumerator kSCTIMER_OutputFallOrMatchEvent

enumerator kSCTIMER_OutputHighOrMatchEvent

enumerator kSCTIMER_OutputLowEvent
enumerator kSCTIMER_OutputRiseEvent
enumerator kSCTIMER_OutputFallEvent
enumerator kSCTIMER_OutputHighEvent
enumerator kSCTIMER_OutputLowAndMatchEvent
enumerator kSCTIMER_OutputRiseAndMatchEvent
enumerator kSCTIMER_OutputFallAndMatchEvent
enumerator kSCTIMER_OutputHighAndMatchEvent

enum _sctimer_interrupt_enable

List of SCTimer interrupts.

Values:

enumerator kSCTIMER_Event0InterruptEnable
Event 0 interrupt
enumerator kSCTIMER_Event1InterruptEnable
Event 1 interrupt
enumerator kSCTIMER_Event2InterruptEnable
Event 2 interrupt
enumerator kSCTIMER_Event3InterruptEnable
Event 3 interrupt
enumerator kSCTIMER_Event4InterruptEnable
Event 4 interrupt
enumerator kSCTIMER_Event5InterruptEnable
Event 5 interrupt
enumerator kSCTIMER_Event6InterruptEnable
Event 6 interrupt
enumerator kSCTIMER_Event7InterruptEnable
Event 7 interrupt
enumerator kSCTIMER_Event8InterruptEnable
Event 8 interrupt
enumerator kSCTIMER_Event9InterruptEnable
Event 9 interrupt
enumerator kSCTIMER_Event10InterruptEnable
Event 10 interrupt
enumerator kSCTIMER_Event11InterruptEnable
Event 11 interrupt
enumerator kSCTIMER_Event12InterruptEnable
Event 12 interrupt

enum _sctimer_status_flags

List of SCTimer flags.

Values:

enumerator kSCTIMER_Event0Flag

Event 0 Flag

enumerator kSCTIMER_Event1Flag

Event 1 Flag

enumerator kSCTIMER_Event2Flag

Event 2 Flag

enumerator kSCTIMER_Event3Flag

Event 3 Flag

enumerator kSCTIMER_Event4Flag

Event 4 Flag

enumerator kSCTIMER_Event5Flag

Event 5 Flag

enumerator kSCTIMER_Event6Flag

Event 6 Flag

enumerator kSCTIMER_Event7Flag

Event 7 Flag

enumerator kSCTIMER_Event8Flag

Event 8 Flag

enumerator kSCTIMER_Event9Flag

Event 9 Flag

enumerator kSCTIMER_Event10Flag

Event 10 Flag

enumerator kSCTIMER_Event11Flag

Event 11 Flag

enumerator kSCTIMER_Event12Flag

Event 12 Flag

enumerator kSCTIMER_BusErrorLFlag

Bus error due to write when L counter was not halted

enumerator kSCTIMER_BusErrorHFlag

Bus error due to write when H counter was not halted

typedef enum *_sctimer_pwm_mode* sctimer_pwm_mode_t

SCTimer PWM operation modes.

typedef enum *_sctimer_counter* sctimer_counter_t

SCTimer counters type.

typedef enum *_sctimer_input* sctimer_input_t

List of SCTimer input pins.

typedef enum *_sctimer_out* sctimer_out_t

List of SCTimer output pins.

typedef enum *_sctimer_pwm_level_select* sctimer_pwm_level_select_t

SCTimer PWM output pulse mode: high-true, low-true or no output.

typedef struct *_sctimer_pwm_signal_param* sctimer_pwm_signal_param_t

Options to configure a SCTimer PWM signal.

```
typedef enum _sctimer_clock_mode sctimer__clock_mode_t
```

SCTimer clock mode options.

```
typedef enum _sctimer_clock_select sctimer__clock_select_t
```

SCTimer clock select options.

```
typedef enum _sctimer_conflict_resolution sctimer__conflict_resolution_t
```

SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

```
typedef enum _sctimer_event_active_direction sctimer__event_active_direction_t
```

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

```
typedef enum _sctimer_event sctimer__event_t
```

List of SCTimer event types.

```
typedef void (*sctimer__event_callback_t)(void)
```

SCTimer callback typedef.

```
typedef enum _sctimer_interrupt_enable sctimer__interrupt_enable_t
```

List of SCTimer interrupts.

```
typedef enum _sctimer_status_flags sctimer__status_flags_t
```

List of SCTimer flags.

```
typedef struct _sctimer_config sctimer__config_t
```

SCTimer configuration structure.

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

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

```
SCT_EV_STATE_STATEMSK(x)
```

```
struct _sctimer_pwm_signal_param
```

#include <fsl_sctimer.h> Options to configure a SCTimer PWM signal.

Public Members

```
sctimer_out_t output
```

The output pin to use to generate the PWM signal

```
sctimer_pwm_level_select_t level
```

PWM output active level select.

```
uint8_t dutyCyclePercent
```

PWM pulse width, value should be between 0 to 100 0 = always inactive signal (0% duty cycle) 100 = always active signal (100% duty cycle).

```
struct _sctimer_config
```

#include <fsl_sctimer.h> SCTimer configuration structure.

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

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

Public Members

`bool enableCounterUnify`

true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters. User can use the 16-bit low counter and the 16-bit high counters at the same time; for Hardware limit, user can not use unified 32-bit counter and any 16-bit low/high counter at the same time.

`sctimer_clock_mode_t clockMode`

SCT clock mode value

`sctimer_clock_select_t clockSelect`

SCT clock select value

`bool enableBidirection_l`

true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter

`bool enableBidirection_h`

true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter. This field is used only if the enableCounterUnify is set to false

`uint8_t prescale_l`

Prescale value to produce the L or unified counter clock

`uint8_t prescale_h`

Prescale value to produce the H counter clock. This field is used only if the enableCounterUnify is set to false

`uint8_t outInitState`

Defines the initial output value

`uint8_t inputsync`

SCT INSYNC value, INSYNC field in the CONFIG register, from bit9 to bit 16. it is used to define synchronization for input N: bit 9 = input 0 bit 10 = input 1 bit 11 = input 2 bit 12 = input 3 All other bits are reserved (bit13 ~bit 16). How User to set the the value for the member inputsync. IE: delay for input0, and input 1, bypasses for input 2 and input 3 MACRO definition in user level. #define INPUTSYNCO (0U) #define INPUTSYNC1 (1U) #define INPUTSYNC2 (2U) #define INPUTSYNC3 (3U) User Code. sctimerInfo.inputsync = (1 « INPUTSYNC2) | (1 « INPUTSYNC3);

2.30 SPI: Serial Peripheral Interface Driver

2.31 SPI DMA Driver

`status_t SPI_MasterTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_dma_callback_t callback, void *userData, dma_handle_t *txHandle, dma_handle_t *rxHandle)`

Initialize the SPI master DMA handle.

This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.

- `handle` – SPI handle pointer.
- `callback` – User callback function called at the end of a transfer.
- `userData` – User data for callback.
- `txHandle` – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- `rxHandle` – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
status_t SPI_MasterTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call `SPI_GetTransferStatus` to poll the transfer status to check whether SPI transfer finished.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI DMA handle pointer.
- `xfer` – Pointer to dma transfer structure.

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_SPI_Busy` – SPI is not idle, is running another transfer.

```
status_t SPI_MasterHalfDuplexTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_half_duplex_transfer_t *xfer)
```

Transfers a block of data using a DMA method.

This function using polling way to do the first half transimission and using DMA way to do the srcond half transimission, the transfer mechanism is half-duplex. When do the second half transimission, code will return right away. When all data is transferred, the callback function is called.

Parameters

- `base` – SPI base pointer
- `handle` – A pointer to the `spi_master_dma_handle_t` structure which stores the transfer state.
- `xfer` – A pointer to the `spi_half_duplex_transfer_t` structure.

Returns

status of `status_t`.

```
static inline status_t SPI_SlaveTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_dma_callback_t callback, void *userData, dma_handle_t *txHandle, dma_handle_t *rxHandle)
```

Initialize the SPI slave DMA handle.

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
static inline status_t SPI_SlaveTransferDMA(SPI_Type *base, spi_dma_handle_t *handle,  
                                           spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

```
void SPI_MasterTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)
```

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

```
status_t SPI_MasterTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t *handle, size_t  
                                       *count)
```

Gets the master DMA transfer remaining bytes.

This function gets the master DMA transfer remaining bytes.

Parameters

- base – SPI peripheral base address.
- handle – A pointer to the spi_dma_handle_t structure which stores the transfer state.
- count – A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

```
static inline void SPI_SlaveTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)
```

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

```
static inline status_t SPI_SlaveTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t
                                                    *handle, size_t *count)
```

Gets the slave DMA transfer remaining bytes.

This function gets the slave DMA transfer remaining bytes.

Parameters

- base – SPI peripheral base address.
- handle – A pointer to the *spi_dma_handle_t* structure which stores the transfer state.
- count – A number of bytes transferred by the non-blocking transaction.

Returns

status of *status_t*.

FSL_SPI_DMA_DRIVER_VERSION

SPI DMA driver version 2.1.1.

```
typedef struct _spi_dma_handle spi_dma_handle_t
```

```
typedef void (*spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status,
void *userData)
```

SPI DMA callback called at the end of transfer.

```
struct _spi_dma_handle
```

#include <fsl_spi_dma.h> SPI DMA transfer handle, users should not touch the content of the handle.

Public Members

```
volatile bool txInProgress
```

Send transfer finished

```
volatile bool rxInProgress
```

Receive transfer finished

```
uint8_t bytesPerFrame
```

Bytes in a frame for SPI transfer

```
uint8_t lastwordBytes
```

The Bytes of lastword for master

```
dma_handle_t *txHandle
```

DMA handler for SPI send

```
dma_handle_t *rxHandle
```

DMA handler for SPI receive

```
spi_dma_callback_t callback
```

Callback for SPI DMA transfer

```
void *userData
```

User Data for SPI DMA callback

```
uint32_t state
```

Internal state of SPI DMA transfer

`size_t` `transferSize`
Bytes need to be transfer

`uint32_t` `instance`
Index of SPI instance

`const uint8_t *``txNextData`
The pointer of next time tx data

`const uint8_t *``txEndData`
The pointer of end of data

`uint8_t *``rxNextData`
The pointer of next time rx data

`uint8_t *``rxEndData`
The pointer of end of rx data

`uint32_t` `dataBytesEveryTime`
Bytes in a time for DMA transfer, default is `DMA_MAX_TRANSFER_COUNT`

2.32 SPI Driver

`FSL_SPI_DRIVER_VERSION`

SPI driver version.

`enum _spi_xfer_option`

SPI transfer option.

Values:

`enumerator` `kSPI_FrameDelay`

A delay may be inserted, defined in the DLY register.

`enumerator` `kSPI_FrameAssert`

SSEL will be deasserted at the end of a transfer

`enum _spi_shift_direction`

SPI data shifter direction options.

Values:

`enumerator` `kSPI_MsbFirst`

Data transfers start with most significant bit.

`enumerator` `kSPI_LsbFirst`

Data transfers start with least significant bit.

`enum _spi_clock_polarity`

SPI clock polarity configuration.

Values:

`enumerator` `kSPI_ClockPolarityActiveHigh`

Active-high SPI clock (idles low).

`enumerator` `kSPI_ClockPolarityActiveLow`

Active-low SPI clock (idles high).

enum _spi_clock_phase

SPI clock phase configuration.

Values:

enumerator kSPI_ClockPhaseFirstEdge

First edge on SCK occurs at the middle of the first cycle of a data transfer.

enumerator kSPI_ClockPhaseSecondEdge

First edge on SCK occurs at the start of the first cycle of a data transfer.

enum _spi_txfifo_watermark

txFIFO watermark values

Values:

enumerator kSPI_TxFifo0

SPI tx watermark is empty

enumerator kSPI_TxFifo1

SPI tx watermark at 1 item

enumerator kSPI_TxFifo2

SPI tx watermark at 2 items

enumerator kSPI_TxFifo3

SPI tx watermark at 3 items

enumerator kSPI_TxFifo4

SPI tx watermark at 4 items

enumerator kSPI_TxFifo5

SPI tx watermark at 5 items

enumerator kSPI_TxFifo6

SPI tx watermark at 6 items

enumerator kSPI_TxFifo7

SPI tx watermark at 7 items

enum _spi_rxfifo_watermark

rxFIFO watermark values

Values:

enumerator kSPI_RxFifo1

SPI rx watermark at 1 item

enumerator kSPI_RxFifo2

SPI rx watermark at 2 items

enumerator kSPI_RxFifo3

SPI rx watermark at 3 items

enumerator kSPI_RxFifo4

SPI rx watermark at 4 items

enumerator kSPI_RxFifo5

SPI rx watermark at 5 items

enumerator kSPI_RxFifo6

SPI rx watermark at 6 items

enumerator kSPI_RxFifo7
SPI rx watermark at 7 items
enumerator kSPI_RxFifo8
SPI rx watermark at 8 items

enum _spi_data_width

Transfer data width.

Values:

enumerator kSPI_Data4Bits
4 bits data width

enumerator kSPI_Data5Bits
5 bits data width

enumerator kSPI_Data6Bits
6 bits data width

enumerator kSPI_Data7Bits
7 bits data width

enumerator kSPI_Data8Bits
8 bits data width

enumerator kSPI_Data9Bits
9 bits data width

enumerator kSPI_Data10Bits
10 bits data width

enumerator kSPI_Data11Bits
11 bits data width

enumerator kSPI_Data12Bits
12 bits data width

enumerator kSPI_Data13Bits
13 bits data width

enumerator kSPI_Data14Bits
14 bits data width

enumerator kSPI_Data15Bits
15 bits data width

enumerator kSPI_Data16Bits
16 bits data width

enum _spi_ssel

Slave select.

Values:

enumerator kSPI_Ssel0
Slave select 0

enumerator kSPI_Ssel1
Slave select 1

enumerator kSPI_Ssel2
Slave select 2

enumerator kSPI_Ssel3
Slave select 3

enum _spi_spol
ssel polarity

Values:

enumerator kSPI_Spol0ActiveHigh

enumerator kSPI_Spol1ActiveHigh

enumerator kSPI_Spol3ActiveHigh

enumerator kSPI_SpolActiveAllHigh

enumerator kSPI_SpolActiveAllLow

SPI transfer status.

Values:

enumerator kStatus_SPI_Busy

SPI bus is busy

enumerator kStatus_SPI_Idle

SPI is idle

enumerator kStatus_SPI_Error

SPI error

enumerator kStatus_SPI_BaudrateNotSupport

Baudrate is not support in current clock source

enumerator kStatus_SPI_Timeout

SPI timeout polling status flags.

enum _spi_interrupt_enable
SPI interrupt sources.

Values:

enumerator kSPI_RxLvlIrq

Rx level interrupt

enumerator kSPI_TxLvlIrq

Tx level interrupt

enum _spi_statusflags
SPI status flags.

Values:

enumerator kSPI_TxEmptyFlag

txFifo is empty

enumerator kSPI_TxNotFullFlag

txFifo is not full

enumerator kSPI_RxNotEmptyFlag

rxFIFO is not empty

enumerator kSPI_RxFullFlag

rxFIFO is full

typedef enum *_spi_xfer_option* spi_xfer_option_t

SPI transfer option.

typedef enum *_spi_shift_direction* spi_shift_direction_t

SPI data shifter direction options.

typedef enum *_spi_clock_polarity* spi_clock_polarity_t

SPI clock polarity configuration.

typedef enum *_spi_clock_phase* spi_clock_phase_t

SPI clock phase configuration.

typedef enum *_spi_txfifo_watermark* spi_txfifo_watermark_t

txFIFO watermark values

typedef enum *_spi_rxfifo_watermark* spi_rxfifo_watermark_t

rxFIFO watermark values

typedef enum *_spi_data_width* spi_data_width_t

Transfer data width.

typedef enum *_spi_ssel* spi_ssel_t

Slave select.

typedef enum *_spi_spol* spi_spol_t

ssel polarity

typedef struct *_spi_delay_config* spi_delay_config_t

SPI delay time configure structure. Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maximum value of these delay time is 15.

typedef struct *_spi_master_config* spi_master_config_t

SPI master user configure structure.

typedef struct *_spi_slave_config* spi_slave_config_t

SPI slave user configure structure.

typedef struct *_spi_transfer* spi_transfer_t

SPI transfer structure.

typedef struct *_spi_half_duplex_transfer* spi_half_duplex_transfer_t

SPI half-duplex(master only) transfer structure.

typedef struct *_spi_config* spi_config_t

Internal configuration structure used in 'spi' and 'spi_dma' driver.

typedef struct *_spi_master_handle* spi_master_handle_t

Master handle type.

typedef *spi_master_handle_t* spi_slave_handle_t

Slave handle type.

typedef void (*spi_master_callback_t)(SPI_Type *base, *spi_master_handle_t* *handle, *status_t* status, void *userData)

SPI master callback for finished transmit.

typedef void (*spi_slave_callback_t)(SPI_Type *base, *spi_slave_handle_t* *handle, *status_t* status, void *userData)

SPI slave callback for finished transmit.

```
typedef void (*flexcomm_spi_master_irq_handler_t)(SPI_Type *base, spi_master_handle_t *handle)
```

Typedef for master interrupt handler.

```
typedef void (*flexcomm_spi_slave_irq_handler_t)(SPI_Type *base, spi_slave_handle_t *handle)
```

Typedef for slave interrupt handler.

```
volatile uint8_t s_dummyData[]
```

SPI default SSEL COUNT.

Global variable for dummy data value setting.

```
SPI_DUMMYDATA
```

SPI dummy transfer data, the data is sent while txBuff is NULL.

```
SPI_RETRY_TIMES
```

Retry times for waiting flag.

```
SPI_DATA(n)
```

```
SPI_CTRLMASK
```

```
SPI_ASSERTNUM_SSEL(n)
```

```
SPI_DEASSERTNUM_SSEL(n)
```

```
SPI_DEASSERT_ALL
```

```
SPI_FIFOWR_FLAGS_MASK
```

```
SPI_FIFOTRIG_TXLVL_GET(base)
```

```
SPI_FIFOTRIG_RXLVL_GET(base)
```

```
struct __spi_delay_config
```

#include <fsl_spi.h> SPI delay time configure structure. Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maximum value of these delay time is 15.

Public Members

```
uint8_t preDelay
```

Delay between SSEL assertion and the beginning of transfer.

```
uint8_t postDelay
```

Delay between the end of transfer and SSEL deassertion.

```
uint8_t frameDelay
```

Delay between frame to frame.

```
uint8_t transferDelay
```

Delay between transfer to transfer.

```
struct __spi_master_config
```

#include <fsl_spi.h> SPI master user configure structure.

Public Members

```
bool enableLoopback
```

Enable loopback for test purpose

`bool enableMaster`
 Enable SPI at initialization time

`spi_clock_polarity_t polarity`
 Clock polarity

`spi_clock_phase_t phase`
 Clock phase

`spi_shift_direction_t direction`
 MSB or LSB

`uint32_t baudRate_Bps`
 Baud Rate for SPI in Hz

`spi_data_width_t dataWidth`
 Width of the data

`spi_ssel_t sselNum`
 Slave select number

`spi_spol_t sselPol`
 Configure active CS polarity

`uint8_t txWatermark`
 txFIFO watermark

`uint8_t rxWatermark`
 rxFIFO watermark

`spi_delay_config_t delayConfig`
 Delay configuration.

`struct __spi_slave_config`
 #include <fsl_spi.h> SPI slave user configure structure.

Public Members

`bool enableSlave`
 Enable SPI at initialization time

`spi_clock_polarity_t polarity`
 Clock polarity

`spi_clock_phase_t phase`
 Clock phase

`spi_shift_direction_t direction`
 MSB or LSB

`spi_data_width_t dataWidth`
 Width of the data

`spi_spol_t sselPol`
 Configure active CS polarity

`uint8_t txWatermark`
 txFIFO watermark

`uint8_t rxWatermark`
 rxFIFO watermark

`struct __spi_transfer`
 #include <fsl_spi.h> SPI transfer structure.

Public Members

```
const uint8_t *txData
    Send buffer

uint8_t *rxData
    Receive buffer

uint32_t configFlags
    Additional option to control transfer, spi_xfer_option_t.

size_t dataSize
    Transfer bytes

struct __spi_half_duplex_transfer
    #include <fsl_spi.h> SPI half-duplex(master only) transfer structure.
```

Public Members

```
const uint8_t *txData
    Send buffer

uint8_t *rxData
    Receive buffer

size_t txDataSize
    Transfer bytes for transmit

size_t rxDataSize
    Transfer bytes

uint32_t configFlags
    Transfer configuration flags, spi_xfer_option_t.

bool isPcsAssertInTransfer
    If PCS pin keep assert between transmit and receive. true for assert and false for de-assert.

bool isTransmitFirst
    True for transmit first and false for receive first.

struct __spi_config
    #include <fsl_spi.h> Internal configuration structure used in 'spi' and 'spi_dma' driver.

struct __spi_master_handle
    #include <fsl_spi.h> SPI transfer handle structure.
```

Public Members

```
const uint8_t *volatile txData
    Transfer buffer

uint8_t *volatile rxData
    Receive buffer

volatile size_t txRemainingBytes
    Number of data to be transmitted [in bytes]

volatile size_t rxRemainingBytes
    Number of data to be received [in bytes]
```

`volatile int8_t toReceiveCount`

The number of data expected to receive in data width. Since the received count and sent count should be the same to complete the transfer, if the sent count is x and the received count is y, toReceiveCount is x-y.

`size_t totalByteCount`

A number of transfer bytes

`volatile uint32_t state`

SPI internal state

`spi_master_callback_t callback`

SPI callback

`void *userData`

Callback parameter

`uint8_t dataWidth`

Width of the data [Valid values: 1 to 16]

`uint8_t sselNum`

Slave select number to be asserted when transferring data [Valid values: 0 to 3]

`uint32_t configFlags`

Additional option to control transfer

`uint8_t txWatermark`

txFIFO watermark

`uint8_t rxWatermark`

rxFIFO watermark

2.33 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.34 USART DMA Driver

```
status_t USART__TransferCreateHandleDMA(USART_Type *base, usart_dma_handle_t *handle,  
                                         usart_dma_transfer_callback_t callback, void  
                                         *userData, dma_handle_t *txDmaHandle,  
                                         dma_handle_t *rxDmaHandle)
```

Initializes the USART handle which is used in transactional functions.

Parameters

- base – USART peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- callback – Callback function.
- userData – User data.
- txDmaHandle – User-requested DMA handle for TX DMA transfer.
- rxDmaHandle – User-requested DMA handle for RX DMA transfer.

```
status_t USART_TransferSendDMA(USART_Type *base, usart_dma_handle_t *handle,  
                               usart_transfer_t *xfer)
```

Sends data using DMA.

This function sends data using DMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_TxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
status_t USART_TransferReceiveDMA(USART_Type *base, usart_dma_handle_t *handle,  
                                  usart_transfer_t *xfer)
```

Receives data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – USART peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_RxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
void USART_TransferAbortSendDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the sent data using DMA.
```

This function aborts send data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure

```
void USART_TransferAbortReceiveDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the received data using DMA.
```

This function aborts the received data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure

```
status_t USART_TransferGetReceiveCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                           uint32_t *count)
```

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
status_t USART_TransferGetSendCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                        uint32_t *count)
```

Get the number of bytes that have been sent.

This function gets the number of bytes that have been sent.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Sent bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
FSL_USART_DMA_DRIVER_VERSION
```

USART dma driver version.

```
typedef struct _usart_dma_handle usart_dma_handle_t
```

```
typedef void (*usart_dma_transfer_callback_t)(USART_Type *base, usart_dma_handle_t *handle,
status_t status, void *userData)
```

UART transfer callback function.

```
struct _usart_dma_handle
```

```
#include <fsl_usart_dma.h> UART DMA handle.
```

Public Members

```
USART_Type *base
```

UART peripheral base address.

```
usart_dma_transfer_callback_t callback
```

Callback function.

```
void *userData
```

UART callback function parameter.

`size_t rxDataSizeAll`

Size of the data to receive.

`size_t txDataSizeAll`

Size of the data to send out.

`dma_handle_t *txDmaHandle`

The DMA TX channel used.

`dma_handle_t *rxDmaHandle`

The DMA RX channel used.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.35 USART Driver

`status_t USART_Init(USART_Type *base, const usart_config_t *config, uint32_t srcClock_Hz)`

Initializes a USART instance with user configuration structure and peripheral clock.

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the `USART_GetDefaultConfig()` function. Example below shows how to use this API to configure USART.

```
usart_config_t usartConfig;
usartConfig.baudRate_Bps = 115200U;
usartConfig.parityMode = kUSART_ParityDisabled;
usartConfig.stopBitCount = kUSART_OneStopBit;
USART_Init(USART1, &usartConfig, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `config` – Pointer to user-defined configuration structure.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_InvalidArgument` – USART base address is not valid
- `kStatus_Success` – Status USART initialize succeed

`void USART_Deinit(USART_Type *base)`

Deinitializes a USART instance.

This function waits for TX complete, disables TX and RX, and disables the USART clock.

Parameters

- `base` – USART peripheral base address.

`void USART_GetDefaultConfig(usart_config_t *config)`

Gets the default configuration structure.

This function initializes the USART configuration structure to a default value. The default values are: `usartConfig->baudRate_Bps = 115200U`; `usartConfig->parityMode = kUSART_ParityDisabled`; `usartConfig->stopBitCount = kUSART_OneStopBit`; `usartConfig->bitCountPerChar = kUSART_8BitsPerChar`; `usartConfig->loopback = false`; `usartConfig->enableTx = false`; `usartConfig->enableRx = false`;

Parameters

- `config` – Pointer to configuration structure.

`status_t USART_SetBaudRate(USART_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)`

Sets the USART instance baud rate.

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the `USART_Init`.

```
USART_SetBaudRate(USART1, 115200U, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `baudrate_Bps` – USART baudrate to be set.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

`status_t USART_Enable32kMode(USART_Type *base, uint32_t baudRate_Bps, bool enableMode32k, uint32_t srcClock_Hz)`

Enable 32 kHz mode which USART uses clock from the RTC oscillator as the clock source.

Please note that in order to use a 32 kHz clock to operate USART properly, the RTC oscillator and its 32 kHz output must be manually enabled by user, by calling `RTC_Init` and setting `SYSCON_RTCOSCTRL_EN` bit to 1. And in 32kHz clocking mode the USART can only work at 9600 baudrate or at the baudrate that 9600 can evenly divide, eg: 4800, 3200.

Parameters

- `base` – USART peripheral base address.
- `baudRate_Bps` – USART baudrate to be set..
- `enableMode32k` – true is 32k mode, false is normal mode.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

`void USART_Enable9bitMode(USART_Type *base, bool enable)`

Enable 9-bit data mode for USART.

This function set the 9-bit mode for USART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- `base` – USART peripheral base address.
- `enable` – true to enable, false to disable.

`static inline void USART_SetMatchAddress(USART_Type *base, uint8_t address)`

Set the USART slave address.

This function configures the address for USART module that works as slave in 9-bit data mode. When the address detection is enabled, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer, otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any USART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- `base` – USART peripheral base address.
- `address` – USART slave address.

`static inline void USART_EnableMatchAddress(USART_Type *base, bool match)`

Enable the USART match address feature.

Parameters

- `base` – USART peripheral base address.
- `match` – true to enable match address, false to disable.

`static inline uint32_t USART_GetStatusFlags(USART_Type *base)`

Get USART status flags.

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators `_usart_flags`. To check a specific status, compare the return value with enumerators in `_usart_flags`. For example, to check whether the TX is empty:

```
if (kUSART_TxFifoNotFullFlag & USART_GetStatusFlags(USART1))
{
    ...
}
```

Parameters

- `base` – USART peripheral base address.

Returns

USART status flags which are ORed by the enumerators in the `_usart_flags`.

`static inline void USART_ClearStatusFlags(USART_Type *base, uint32_t mask)`

Clear USART status flags.

This function clear supported USART status flags. The mask is a logical OR of enumeration members. See `kUSART_AllClearFlags`. For example:

```
USART_ClearStatusFlags(USART1, kUSART_TxError | kUSART_RxError)
```

Parameters

- base – USART peripheral base address.
- mask – status flags to be cleared.

```
static inline void USART_EnableInterrupts(USART_Type *base, uint32_t mask)
```

Enables USART interrupts according to the provided mask.

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. For example, to enable TX empty interrupt and RX full interrupt:

```
USART_EnableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART_
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_usart_interrupt_enable`.

```
static inline void USART_DisableInterrupts(USART_Type *base, uint32_t mask)
```

Disables USART interrupts according to a provided mask.

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
USART_DisableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART_
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to disable. Logical OR of `_usart_interrupt_enable`.

```
static inline uint32_t USART_GetEnabledInterrupts(USART_Type *base)
```

Returns enabled USART interrupts.

This function returns the enabled USART interrupts.

Parameters

- base – USART peripheral base address.

```
static inline void USART_EnableTxDMA(USART_Type *base, bool enable)
```

Enable DMA for Tx.

```
static inline void USART_EnableRxDMA(USART_Type *base, bool enable)
```

Enable DMA for Rx.

```
static inline void USART_EnableCTS(USART_Type *base, bool enable)
```

Enable CTS. This function will determine whether CTS is used for flow control.

Parameters

- base – USART peripheral base address.
- enable – Enable CTS or not, true for enable and false for disable.

static inline void USART__EnableContinuousSCLK(USART_Type *base, bool enable)

Continuous Clock generation. By default, SCLK is only output while data is being transmitted in synchronous mode. Enable this function, SCLK will run continuously in synchronous mode, allowing characters to be received on Un_RxD independently from transmission on Un_TXD).

Parameters

- base – USART peripheral base address.
- enable – Enable Continuous Clock generation mode or not, true for enable and false for disable.

static inline void USART__EnableAutoClearSCLK(USART_Type *base, bool enable)

Enable Continuous Clock generation bit auto clear. While enable this function, the Continuous Clock bit is automatically cleared when a complete character has been received. This bit is cleared at the same time.

Parameters

- base – USART peripheral base address.
- enable – Enable auto clear or not, true for enable and false for disable.

static inline void USART__SetRxFifoWatermark(USART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Rx FIFO watermark.

static inline void USART__SetTxFifoWatermark(USART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Tx FIFO watermark.

static inline void USART__WriteByte(USART_Type *base, uint8_t data)

Writes to the FIFOWR register.

This function writes data to the txFIFO directly. The upper layer must ensure that txFIFO has space for data to write before calling this function.

Parameters

- base – USART peripheral base address.
- data – The byte to write.

static inline uint8_t USART__ReadByte(USART_Type *base)

Reads the FIFORD register directly.

This function reads data from the rxFIFO directly. The upper layer must ensure that the rxFIFO is not empty before calling this function.

Parameters

- base – USART peripheral base address.

Returns

The byte read from USART data register.

```
static inline uint8_t USART_GetRxFifoCount(USART_Type *base)
```

Gets the rx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

rx FIFO data count.

```
static inline uint8_t USART_GetTxFifoCount(USART_Type *base)
```

Gets the tx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

tx FIFO data count.

```
void USART_SendAddress(USART_Type *base, uint8_t address)
```

Transmit an address frame in 9-bit data mode.

Parameters

- base – USART peripheral base address.
- address – USART slave address.

```
status_t USART_WriteBlocking(USART_Type *base, const uint8_t *data, size_t length)
```

Writes to the TX register using a blocking method.

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Parameters

- base – USART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_USART_Timeout – Transmission timed out and was aborted.
- kStatus_InvalidArgument – Invalid argument.
- kStatus_Success – Successfully wrote all data.

```
status_t USART_ReadBlocking(USART_Type *base, uint8_t *data, size_t length)
```

Read RX data register using a blocking method.

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data and read data from the TX register.

Parameters

- base – USART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_USART_FramingError – Receiver overrun happened while receiving data.
- kStatus_USART_ParityError – Noise error happened while receiving data.

- `kStatus_USART_NoiseError` – Framing error happened while receiving data.
- `kStatus_USART_RxError` – Overflow or underflow rxFIFO happened.
- `kStatus_USART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

`status_t` `USART_TransferCreateHandle`(`USART_Type` *base, `usart_handle_t` *handle, `usart_transfer_callback_t` callback, void *userData)

Initializes the USART handle.

This function initializes the USART handle which can be used for other USART transactional APIs. Usually, for a specified USART instance, call this API once to get the initialized handle.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

`status_t` `USART_TransferSendNonBlocking`(`USART_Type` *base, `usart_handle_t` *handle, `usart_transfer_t` *xfer)

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the IRQ handler, the USART driver calls the callback function and passes the `kStatus_USART_TxIdle` as status parameter.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_USART_TxBusy` – Previous transmission still not finished, data not all written to TX register yet.
- `kStatus_InvalidArgument` – Invalid argument.

void `USART_TransferStartRingBuffer`(`USART_Type` *base, `usart_handle_t` *handle, uint8_t *ringBuffer, size_t ringBufferSize)

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the `USART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

- base – USART peripheral base address.

- `handle` – USART handle pointer.
- `ringBuffer` – Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
- `ringBufferSize` – size of the ring buffer.

`void USART_TransferStopRingBuffer(USART_Type *base, usart_handle_t *handle)`

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

`size_t USART_TransferGetRxRingBufferLength(usart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `handle` – USART handle pointer.

Returns

Length of received data in RX ring buffer.

`void USART_TransferAbortSend(USART_Type *base, usart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are still not sent out.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

`status_t USART_TransferGetSendCount(USART_Type *base, usart_handle_t *handle, uint32_t *count)`

Get the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by interrupt method.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

`status_t USART_TransferReceiveNonBlocking(USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer, size_t *receivedBytes)`

Receives a buffer of data using an interrupt method.

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the USART driver. When the new

data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_USART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the USART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.
- `xfer` – USART transfer structure, see `usart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into transmit queue.
- `kStatus_USART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void USART_TransferAbortReceive(USART_Type *base, usart_handle_t *handle)`

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to find out how many bytes not received yet.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

`status_t USART_TransferGetReceiveCount(USART_Type *base, usart_handle_t *handle, uint32_t *count)`

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.
- `count` – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`void USART_TransferHandleIRQ(USART_Type *base, usart_handle_t *handle)`

USART IRQ handle function.

This function handles the USART transmit and receive IRQ request.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

FSL_USART_DRIVER_VERSION

USART driver version.

Error codes for the USART driver.

Values:

enumerator kStatus_USART_TxBusy
Transmitter is busy.

enumerator kStatus_USART_RxBusy
Receiver is busy.

enumerator kStatus_USART_TxIdle
USART transmitter is idle.

enumerator kStatus_USART_RxIdle
USART receiver is idle.

enumerator kStatus_USART_TxError
Error happens on txFIFO.

enumerator kStatus_USART_RxError
Error happens on rxFIFO.

enumerator kStatus_USART_RxRingBufferOverrun
Error happens on rx ring buffer

enumerator kStatus_USART_NoiseError
USART noise error.

enumerator kStatus_USART_FramingError
USART framing error.

enumerator kStatus_USART_ParityError
USART parity error.

enumerator kStatus_USART_BaudrateNotSupport
Baudrate is not support in current clock source

enum _usart_sync_mode
USART synchronous mode.

Values:

enumerator kUSART_SyncModeDisabled
Asynchronous mode.

enumerator kUSART_SyncModeSlave
Synchronous slave mode.

enumerator kUSART_SyncModeMaster
Synchronous master mode.

enum _usart_parity_mode
USART parity mode.

Values:

enumerator kUSART_ParityDisabled
Parity disabled

enumerator kUSART_ParityEven

Parity enabled, type even, bit setting: PE|PT = 10

enumerator kUSART_ParityOdd

Parity enabled, type odd, bit setting: PE|PT = 11

enum _usart_stop_bit_count

USART stop bit count.

Values:

enumerator kUSART_OneStopBit

One stop bit

enumerator kUSART_TwoStopBit

Two stop bits

enum _usart_data_len

USART data size.

Values:

enumerator kUSART_7BitsPerChar

Seven bit mode

enumerator kUSART_8BitsPerChar

Eight bit mode

enum _usart_clock_polarity

USART clock polarity configuration, used in sync mode.

Values:

enumerator kUSART_RxSampleOnFallingEdge

Un_RXD is sampled on the falling edge of SCLK.

enumerator kUSART_RxSampleOnRisingEdge

Un_RXD is sampled on the rising edge of SCLK.

enum _usart_txfifo_watermark

txFIFO watermark values

Values:

enumerator kUSART_TxFifo0

USART tx watermark is empty

enumerator kUSART_TxFifo1

USART tx watermark at 1 item

enumerator kUSART_TxFifo2

USART tx watermark at 2 items

enumerator kUSART_TxFifo3

USART tx watermark at 3 items

enumerator kUSART_TxFifo4

USART tx watermark at 4 items

enumerator kUSART_TxFifo5

USART tx watermark at 5 items

enumerator kUSART_TxFifo6

USART tx watermark at 6 items

enumerator kUSART_TxFifo7
USART tx watermark at 7 items

enum __usart_rxfifo_watermark
rxFIFO watermark values

Values:

enumerator kUSART_RxFifo1
USART rx watermark at 1 item

enumerator kUSART_RxFifo2
USART rx watermark at 2 items

enumerator kUSART_RxFifo3
USART rx watermark at 3 items

enumerator kUSART_RxFifo4
USART rx watermark at 4 items

enumerator kUSART_RxFifo5
USART rx watermark at 5 items

enumerator kUSART_RxFifo6
USART rx watermark at 6 items

enumerator kUSART_RxFifo7
USART rx watermark at 7 items

enumerator kUSART_RxFifo8
USART rx watermark at 8 items

enum __usart_interrupt_enable
USART interrupt configuration structure, default settings all disabled.

Values:

enumerator kUSART_TxErrorInterruptEnable

enumerator kUSART_RxErrorInterruptEnable

enumerator kUSART_TxLevelInterruptEnable

enumerator kUSART_RxLevelInterruptEnable

enumerator kUSART_TxIdleInterruptEnable
Transmitter idle.

enumerator kUSART_CtsChangeInterruptEnable
Change in the state of the CTS input.

enumerator kUSART_RxBreakChangeInterruptEnable
Break condition asserted or deasserted.

enumerator kUSART_RxStartInterruptEnable
Rx start bit detected.

enumerator kUSART_FramingErrorInterruptEnable
Framing error detected.

enumerator kUSART_ParityErrorInterruptEnable
Parity error detected.

enumerator kUSART_NoiseErrorInterruptEnable
Noise error detected.

enumerator kUSART_AutoBaudErrorInterruptEnable
Auto baudrate error detected.

enumerator kUSART_AllInterruptEnables

enum __usart_flags

USART status flags.

This provides constants for the USART status flags for use in the USART functions.

Values:

enumerator kUSART_TxError
TXERR bit, sets if TX buffer is error

enumerator kUSART_RxError
RXERR bit, sets if RX buffer is error

enumerator kUSART_TxFifoEmptyFlag
TXEMPTY bit, sets if TX buffer is empty

enumerator kUSART_TxFifoNotFullFlag
TXNOTFULL bit, sets if TX buffer is not full

enumerator kUSART_RxFifoNotEmptyFlag
RXNOEMPTY bit, sets if RX buffer is not empty

enumerator kUSART_RxFifoFullFlag
RXFULL bit, sets if RX buffer is full

enumerator kUSART_RxIdleFlag
Receiver idle.

enumerator kUSART_TxIdleFlag
Transmitter idle.

enumerator kUSART_CtsAssertFlag
CTS signal high.

enumerator kUSART_CtsChangeFlag
CTS signal changed interrupt status.

enumerator kUSART_BreakDetectFlag
Break detected. Self cleared when rx pin goes high again.

enumerator kUSART_BreakDetectChangeFlag
Break detect change interrupt flag. A change in the state of receiver break detection.

enumerator kUSART_RxStartFlag
Rx start bit detected interrupt flag.

enumerator kUSART_FramingErrorFlag
Framing error interrupt flag.

enumerator kUSART_ParityErrorFlag
parity error interrupt flag.

enumerator kUSART_NoiseErrorFlag
Noise error interrupt flag.

enumerator `kUSART_AutobaudErrorFlag`

Auto baudrate error interrupt flag, caused by the baudrate counter timeout before the end of start bit.

enumerator `kUSART_AllClearFlags`

typedef enum `_usart_sync_mode` `usart_sync_mode_t`

USART synchronous mode.

typedef enum `_usart_parity_mode` `usart_parity_mode_t`

USART parity mode.

typedef enum `_usart_stop_bit_count` `usart_stop_bit_count_t`

USART stop bit count.

typedef enum `_usart_data_len` `usart_data_len_t`

USART data size.

typedef enum `_usart_clock_polarity` `usart_clock_polarity_t`

USART clock polarity configuration, used in sync mode.

typedef enum `_usart_txfifo_watermark` `usart_txfifo_watermark_t`

txFIFO watermark values

typedef enum `_usart_rxfifo_watermark` `usart_rxfifo_watermark_t`

rxFIFO watermark values

typedef struct `_usart_config` `usart_config_t`

USART configuration structure.

typedef struct `_usart_transfer` `usart_transfer_t`

USART transfer structure.

typedef struct `_usart_handle` `usart_handle_t`

typedef void (*`usart_transfer_callback_t`)(`USART_Type` *`base`, `usart_handle_t` *`handle`, `status_t` `status`, void *`userData`)

USART transfer callback function.

typedef void (*`flexcomm_usart_irq_handler_t`)(`USART_Type` *`base`, `usart_handle_t` *`handle`)

Typedef for usart interrupt handler.

uint32_t `USART_GetInstance(USART_Type *base)`

Returns instance number for USART peripheral base address.

`USART_FIFOTRIG_TXLVL_GET(base)`

`USART_FIFOTRIG_RXLVL_GET(base)`

`UART_RETRY_TIMES`

Retry times for waiting flag.

Defining to zero means to keep waiting for the flag until it is assert/deassert in blocking transfer; otherwise the program will wait until the `UART_RETRY_TIMES` counts down to 0, if the flag still remains unchanged then program will return `kStatus_USART_Timeout`. It is not advised to use this macro in formal application to prevent any hardware error because the actual wait period is affected by the compiler and optimization.

struct `_usart_config`

`#include <fsl_usart.h>` USART configuration structure.

Public Members

```

uint32_t baudRate_Bps
    USART baud rate

usart_parity_mode_t parityMode
    Parity mode, disabled (default), even, odd

usart_stop_bit_count_t stopBitCount
    Number of stop bits, 1 stop bit (default) or 2 stop bits

usart_data_len_t bitCountPerChar
    Data length - 7 bit, 8 bit

bool loopback
    Enable peripheral loopback

bool enableRx
    Enable RX

bool enableTx
    Enable TX

bool enableContinuousSCLK
    USART continuous Clock generation enable in synchronous master mode.

bool enableMode32k
    USART uses 32 kHz clock from the RTC oscillator as the clock source.

bool enableHardwareFlowControl
    Enable hardware control RTS/CTS

usart_txfifo_watermark_t txWatermark
    txFIFO watermark

usart_rxfifo_watermark_t rxWatermark
    rxFIFO watermark

usart_sync_mode_t syncMode
    Transfer mode select - asynchronous, synchronous master, synchronous slave.

usart_clock_polarity_t clockPolarity
    Selects the clock polarity and sampling edge in synchronous mode.

struct __usart_transfer
    #include <fsl_usart.h> USART transfer structure.

```

Public Members

```

size_t dataSize
    The byte count to be transfer.

struct __usart_handle
    #include <fsl_usart.h> USART handle structure.

```

Public Members

```

const uint8_t *volatile txData
    Address of remaining data to send.

```

`volatile size_t txDataSize`
Size of the remaining data to send.

`size_t txDataSizeAll`
Size of the data to send out.

`uint8_t *volatile rxData`
Address of remaining data to receive.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t rxDataSizeAll`
Size of the data to receive.

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

`volatile uint16_t rxRingBufferTail`
Index for the user to get data from the ring buffer.

`usart_transfer_callback_t callback`
Callback function.

`void *userData`
USART callback function parameter.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state

`uint8_t txWatermark`
txFIFO watermark

`uint8_t rxWatermark`
rxFIFO watermark

`union __unnamed12__`

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.

2.36 UTICK: MictoTick Timer Driver

`void UTICK_Init(UTICK_Type *base)`

Initializes an UTICK by turning its bus clock on.

`void UTICK_Deinit(UTICK_Type *base)`

Deinitializes a UTICK instance.

This function shuts down Utick bus clock

Parameters

- `base` – UTICK peripheral base address.

`uint32_t UTICK_GetStatusFlags(UTICK_Type *base)`

Get Status Flags.

This returns the status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

status register value

`void UTICK_ClearStatusFlags(UTICK_Type *base)`

Clear Status Interrupt Flags.

This clears intr status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

none

`void UTICK_SetTick(UTICK_Type *base, utick_mode_t mode, uint32_t count, utick_callback_t cb)`

Starts UTICK.

This function starts a repeat/onetime countdown with an optional callback

Parameters

- `base` – UTICK peripheral base address.
- `mode` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `count` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `cb` – UTICK callback (can be left as `NULL` if none, otherwise should be a `void func(void)`)

Returns

none

`void UTICK_HandleIRQ(UTICK_Type *base, utick_callback_t cb)`

UTICK Interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in `UTICK_SetTick()`). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- `base` – UTICK peripheral base address.

- `cb` – callback scheduled for this instance of UTICK

Returns

none

FSL_UTICK_DRIVER_VERSION

UTICK driver version 2.0.5.

enum `_utick_mode`

UTICK timer operational mode.

Values:

enumerator `kUTICK_Onetime`

Trigger once

enumerator `kUTICK_Repeat`

Trigger repeatedly

typedef enum `_utick_mode` `utick_mode_t`

UTICK timer operational mode.

typedef void (`*utick_callback_t`)(void)

UTICK callback function.

2.37 WWDT: Windowed Watchdog Timer Driver

void WWDT_GetDefaultConfig(`wwdt_config_t` *config)

Initializes WWDT configure structure.

This function initializes the WWDT configure structure to default value. The default value are:

```
config->enableWwdt = true;
config->enableWatchdogReset = false;
config->enableWatchdogProtect = false;
config->enableLockOscillator = false;
config->windowValue = 0xFFFFFfU;
config->timeoutValue = 0xFFFFFfU;
config->warningValue = 0;
```

See also:

`wwdt_config_t`

Parameters

- `config` – Pointer to WWDT config structure.

void WWDT_Init(WWDT_Type *base, const `wwdt_config_t` *config)

Initializes the WWDT.

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
wwdt_config_t config;
WWDT_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
WWDT_Init(wwdt_base,&config);
```

Parameters

- base – WWDT peripheral base address
- config – The configuration of WWDT

```
void WWDT_Deinit(WWDT_Type *base)
```

Shuts down the WWDT.

This function shuts down the WWDT.

Parameters

- base – WWDT peripheral base address

```
static inline void WWDT_Enable(WWDT_Type *base)
```

Enables the WWDT module.

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

- base – WWDT peripheral base address

```
static inline void WWDT_Disable(WWDT_Type *base)
```

Disables the WWDT module.

Deprecated:

Do not use this function. It will be deleted in next release version, for once the bit field of WDEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

- base – WWDT peripheral base address

```
static inline uint32_t WWDT_GetStatusFlags(WWDT_Type *base)
```

Gets all WWDT status flags.

This function gets all status flags.

Example for getting Timeout Flag:

```
uint32_t status;  
status = WWDT_GetStatusFlags(wwdt_base) & kWWDT_TimeoutFlag;
```

Parameters

- base – WWDT peripheral base address

Returns

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

```
void WWDT_ClearStatusFlags(WWDT_Type *base, uint32_t mask)
```

Clear WWDT flag.

This function clears WWDT status flag.

Example for clearing warning flag:

```
WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
```

Parameters

- `base` – WWDT peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `_wwdt_status_flags_t`

`static inline void WWDT_SetWarningValue(WWDT_Type *base, uint32_t warningValue)`

Set the WWDT warning value.

The WDWARNINT register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by WARNINT, an interrupt will be generated after the subsequent WDCLK.

Parameters

- `base` – WWDT peripheral base address
- `warningValue` – WWDT warning value.

`static inline void WWDT_SetTimeoutValue(WWDT_Type *base, uint32_t timeoutCount)`

Set the WWDT timeout value.

This function sets the timeout value. Every time a feed sequence occurs the value in the TC register is loaded into the Watchdog timer. Writing a value below 0xFF will cause 0xFF to be loaded into the TC register. Thus the minimum time-out interval is $TWDCLK * 256 * 4$. If `enableWatchdogProtect` flag is true in `wwdt_config_t` config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the WDTOF flag.

Parameters

- `base` – WWDT peripheral base address
- `timeoutCount` – WWDT timeout value, count of WWDT clock tick.

`static inline void WWDT_SetWindowValue(WWDT_Type *base, uint32_t windowValue)`

Sets the WWDT window value.

The WINDOW register determines the highest TV value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in WINDOW, a watchdog event will occur. To disable windowing, set `windowValue` to 0xFFFFFFFF (maximum possible timer value) so windowing is not in effect.

Parameters

- `base` – WWDT peripheral base address
- `windowValue` – WWDT window value.

`void WWDT_Refresh(WWDT_Type *base)`

Refreshes the WWDT timer.

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

- `base` – WWDT peripheral base address

`FSL_WWDT_DRIVER_VERSION`

Defines WWDT driver version.

`WWDT_FIRST_WORD_OF_REFRESH`

First word of refresh sequence

`WWDT_SECOND_WORD_OF_REFRESH`

Second word of refresh sequence

enum `_wwdt_status_flags_t`

WWDT status flags.

This structure contains the WWDT status flags for use in the WWDT functions.

Values:

enumerator `kWWDT_TimeoutFlag`

Time-out flag, set when the timer times out

enumerator `kWWDT_WarningFlag`

Warning interrupt flag, set when timer is below the value `WDWARNINT`

typedef struct `_wwdt_config` `wwdt_config_t`

Describes WWDT configuration structure.

struct `_wwdt_config`

#include `<fsl_wwdt.h>` Describes WWDT configuration structure.

Public Members

bool `enableWwdt`

Enables or disables WWDT

bool `enableWatchdogReset`

true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset

bool `enableWatchdogProtect`

true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time

bool `enableLockOscillator`

true: Disabling or powering down the watchdog oscillator is prevented Once set, this bit can only be cleared by a reset false: Do not lock oscillator

uint32_t `windowValue`

Window value, set this to 0xFFFFFFFF if windowing is not in effect

uint32_t `timeoutValue`

Timeout value

uint32_t `warningValue`

Watchdog time counter value that will generate a warning interrupt. Set this to 0 for no warning

uint32_t `clockFreq_Hz`

Watchdog clock source frequency.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called “packet-driven BDM” interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT](<https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/>) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to [FreeMASTER community](#) or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the `FMSTR_Init` function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a “middleware” component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER “middleware” driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample applications Zephyr sample applications demonstrate Kconfig and Device Tree configuration which configure the FreeMASTER middleware module for a selected communication option (Serial, CAN, Network or RTT).

Refer to *readme.md* files in each sample directory for description of configuration options required to implement FreeMASTER connectivity.

Description

This section shows how to add the FreeMASTER Communication Driver into application and how to configure the connection to the FreeMASTER visualization tool.

Features The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space.
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables.
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control an embedded application.

Board Detection The FreeMASTER protocol V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

Memory Read This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must be shorter than the MTU to fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The driver uses size-aligned operations to read the target memory (for example; uses proper read-word instruction when an address is aligned to 4 bytes).

Memory Write Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; uses proper write-word instruction when an address is aligned to 4 bytes).

Masked Memory Write To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

Oscilloscope The protocol and driver enables any number of variables to be read at once with a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

The driver can be configured to support any number of Oscilloscope instances and enable simultaneously running graphs to be displayed on the host computer screen.

Recorder The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

The driver can be configured to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables setting the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

TSA With the TSA feature, data types and variables can be described directly in the application source code. Such information is later provided to the FreeMASTER tool which may use it instead of reading symbol data from the application ELF executable file.

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

TSA Safety When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

Application commands The Application Commands are high-level messages that can be delivered from the PC Host to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing in general.

Pipes The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losses.

Serial single-wire operation The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- “External” single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- “True” single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as “automatic” in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the .c files must be added to the project, compiled, and linked together with the application.
 - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
 - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
 - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
 - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
 - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
 - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.
- **src/drivers/[sdk]/serial** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- **src/drivers/[sdk]/can** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- **src/drivers/[sdk]/network** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
 - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
 - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR    [0|1]
#define FMSTR_SHORT_INTR  [0|1]
#define FMSTR_POLL_DRIVEN [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See [Driver interrupt modes](#).

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- FMSTR_SERIAL - serial communication protocol
- FMSTR_CAN - using CAN communication
- FMSTR_PDBDM - using packet-driven BDM communication
- FMSTR_NET - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as **FMSTR_SERIAL_DRV**. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32...255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0...255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the “True” single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE

```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetCanBaseAddress()` to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with `FMSTR_CAN_EXTID` bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options**FMSTR_DISABLE**

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+©W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options**FMSTR_USE_SCOPE**

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options**FMSTR_USE_RECORDER**

```
#define FMSTR_USE_RECORDER [number]
```

Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()' API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.
Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature. Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using FMSTR_RegisterAppCmdCall(). Default value is 0.

TSA options**FMSTR_USE_TSA**

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool. Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables. Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project. Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions. Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used. Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support. The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_QUEUE_SIZE*) and the character time is the time needed to transmit or receive a single byte over the SCI line.

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

In this mode, both the communication and the FreeMASTER protocol decoding are done in the *FMSTR_Poll* routine. No interrupts are needed and the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, and similar handlers compile to an empty code.

When using this mode, ensure that the *FMSTR_Poll* function is called by the application at least once per the serial “character time” which is the time needed to transmit or receive a single character.

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the *FMSTR_Poll* routine. An application interrupt can occur in the middle of the

Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

The same issue may appear even in the full interrupt mode (FMSTR_LONG_INTR), if volatile variables are modified in the interrupt code with a priority higher than the priority of the communication interrupt.

Data types Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the platform-specific header file.

To prevent name conflicts with the symbols used in the application, all data types, macros, and functions have the FMSTR_ prefix. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

Communication interface initialization The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is the application startup code responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

Note: It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

FreeMASTER Recorder calls When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere else. The example applications provided together with the driver code call the FMSTR_Recorder in the main application loop.

In applications where FMSTR_Recorder is called periodically with a constant period, specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

Driver usage Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all *.c files of the FreeMASTER driver from the `src/common/platforms/[your_platform]` folder are a part of the project. See [Driver files](#) for more details.
- Configure the FreeMASTER driver by creating or editing the `freemaster_cfg.h` file and by saving it into the application project directory. See [Driver configuration](#) for more details.
- Include the `freemaster.h` file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, call the FMSTR_Poll API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally so that the interrupts can be handled by the CPU.

Communication troubleshooting The most common problem that causes communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When a communication between the PC Host running FreeMASTER and the target MCU cannot be established, try enabling the FMSTR_DEBUG_TX option in the `freemaster_cfg.h` file and call the FMSTR_Poll function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Driver API

This section describes the driver Application Programmers' Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

Control API There are three key functions to initialize and use the driver.

FMSTR_Init

Prototype

```
FMSTR_BOOL FMSTR_Init(void);
```

- Declaration: `freemaster.h`
- Implementation: `freemaster_protocol.c`

Description This function initializes the internal variables of the FreeMASTER driver and enables the communication interface. This function does not change the configuration of the selected communication module. The hardware module must be initialized before the *FMSTR_Init* function is called.

A call to this function must occur before calling any other FreeMASTER driver API functions.

FMSTR_Poll

Prototype

```
void FMSTR_Poll(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see *Driver interrupt modes*). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the *FMSTR_Poll* function is called during the “idle” time in the main application task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the *FMSTR_Poll* function is called at least once per the time calculated as:

$$N * Tchar$$

where:

- *N* is equal to the length of the receive FIFO queue (configured by the *FMSTR_COMM_RQUEUE_SIZE* macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

Note: In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void);
void FMSTR_CanIsr(void);
```

- Declaration: *freemaster.h*
- Implementation: *hw-specific low-level driver C file*

Description This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see *Driver interrupt modes*), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

Note: In a poll-driven mode, this function is compiled as an empty function and does not have to be used.

Recorder API

FMSTR_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance 0 which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the *FMSTR_REC_BUFF_SIZE* and *FMSTR_REC_TIMEBASE* options.

For more information, see [Configurable items](#).

FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance *recIndex*. If the selected Recorder is not active when the *FMSTR_Recorder* function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down the follow-up samples (number of *FMSTR_Recorder* function calls) and de-activates the Recorder when the required post-trigger samples are finished.

The *FMSTR_Recorder* function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

FMSTR_RecorderTrigger

Prototype

```
void FMSTR_RecorderTrigger(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application code for programmatic control over the Recorder triggering. This can be useful when a more complex triggering conditions need to be used.

Fast Recorder API The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

TSA Tables When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), it defines the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always exactly one TSA Table List defined, which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. The symbols can then be used to create FreeMASTER variables for visualization or control.

TSA table definition The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide an access to the individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-language symbol.

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

After this opening macro, the TSA descriptors are placed using these macros:

```
/* Adding variable descriptors */
FMSTR_TSA_RW_VAR(name, type) /* read/write variable entry */
FMSTR_TSA_RO_VAR(name, type) /* read-only variable entry */

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
FMSTR_TSA_MEMBER(struct_name, member_name, type) /* structure member entry */

/* Memory blocks */
FMSTR_TSA_RW_MEM(name, type, address, size) /* read/write memory block */
FMSTR_TSA_RO_MEM(name, type, address, size) /* read-only memory block */
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

TSA descriptor parameters The TSA descriptor macros accept these parameters:

- *name* — variable name. The variable must be defined before the TSA descriptor references it.
- *type* — variable or member type. Only one of the pre-defined type constants may be used (see below).
- *struct_name* — structure type name. The type must be defined (typedef) before the TSA descriptor references it.

- *member_name* — structure member name.

Note: The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table.

Note: To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

TSA variable types The table lists *type* identifiers which can be used in TSA descriptors:

Constant	Description
FMSTR_TSA_UINTn	Unsigned integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size <i>n</i> bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(<i>m,n</i>)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(<i>m,n</i>)	Unsigned fractional number in general UQ form (m+n total bits)
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type
FMSTR_TSA_POINTER	Generic pointer type defined (platform-specific 16 or 32 bit)
FM-STR_TSA_USERTYPE(<i>name</i>)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()
```

```
FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *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 **coremqtt-agent**

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

Readme

4.1.8 **corepkcs11**

PKCS #11 key management library.

Readme

4.1.9 **freertos-plus-tcp**

Open source RTOS FreeRTOS Plus TCP.

Readme