MCUXpresso SDK Documentation
Release 25.12.00

Table of contents

This documentation contains information specific to the lpcxpresso54s018 board.

Chapter 1

LPCXpresso54S018

1.1   Overview

The LPCXpresso54S018 board provides a powerful and flexible platform for easy evaluation of the LPC54S00 family of microcontrollers. This board is an LPCXpresso V3 style, the latest generation of the original and highly successful LPCXpresso board family. These boards provide Arduino UNO compatible shield connectors with additional expansion ports around the Arduino footprint and also a PMod/host interface port. These boards feature an on-board LPC-Link2 debug probe based on the LPC4322 MCU for a performance debug experience over high speed USB, with easy firmware update options to support CMSIS-DSP or a special version of J-link OBD from SEGGER. The board can also be used with an external debug probe such as those from SEGGER and P&E.

The LPC54S00 series is fully supported by NXPs MCUXpresso suite of free software and tools, which include an Eclipse-based IDE, configuration tools and extensive SDK drivers/examples available at https://mcuxpresso.nxp.com. MCUXpresso SDK includes project files for use with IDEs from lead partners Keil and IAR, and these IDEs are also fully supported by the MCUXpresso pin, clock and peripheral configuration tools.

The LPCXpresso54S018 board includes an extensive set of connectors and on-board peripherals to enable full evaluation of the target MCU and development of highly functional prototypes. The on-board peripherals are complemented by a set of drivers and examples in the MCUXpresso SDK.

MCU device and part on board is shown below:
- Device: LPC54S018
- PartNumber: LPC54S018JET180

1.2   Getting Started with MCUXpresso SDK Package

1.2.1   Getting Started with MCUXpresso SDK Package

Starting with version 25.09.00, MCUXpresso SDK introduced two package versions for offline development:

- **Classic SDK Package**: Traditional board-specific packages with pre-configured IDE projects for MCUXpresso IDE, IAR, Keil, and other toolchains.

- **Repository-Layout SDK Package**: Board-specific packages that maintain the same structure and build system as the GitHub Repository SDK, providing offline access to the repository SDK development experience. Available when selecting the ARMGCC toolchain.

From version 25.12.00 onward:

- When you select ARMGCC, the SDK download will use the Repository-Layout version.

- For all other toolchains, the SDK download will remain in the Classic version.

Note: The Repository-Layout SDK package was first introduced in version 25.09.00, but initially only for MCXW23x platforms.

Classic SDK Package

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

For supported toolchain versions, see *MCUXpresso SDK Release Notes* (document MCUXSDKRN).

For more details about MCUXpresso SDK, see MCUXpresso Software Development Kit (SDK).
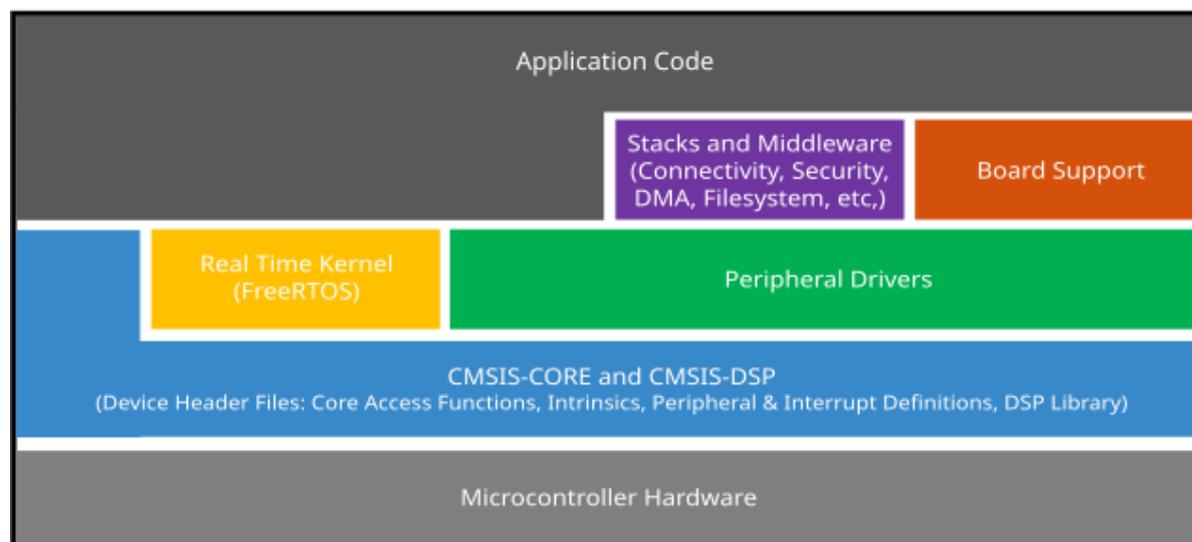
MCUXpresso SDK board support package folders MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top-level boards folder and each supported board has its own folder (an MCUXpresso SDK package can support multiple boards). Within each <board_name> folder, there are various subfolders to classify the type of examples it contains. 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:
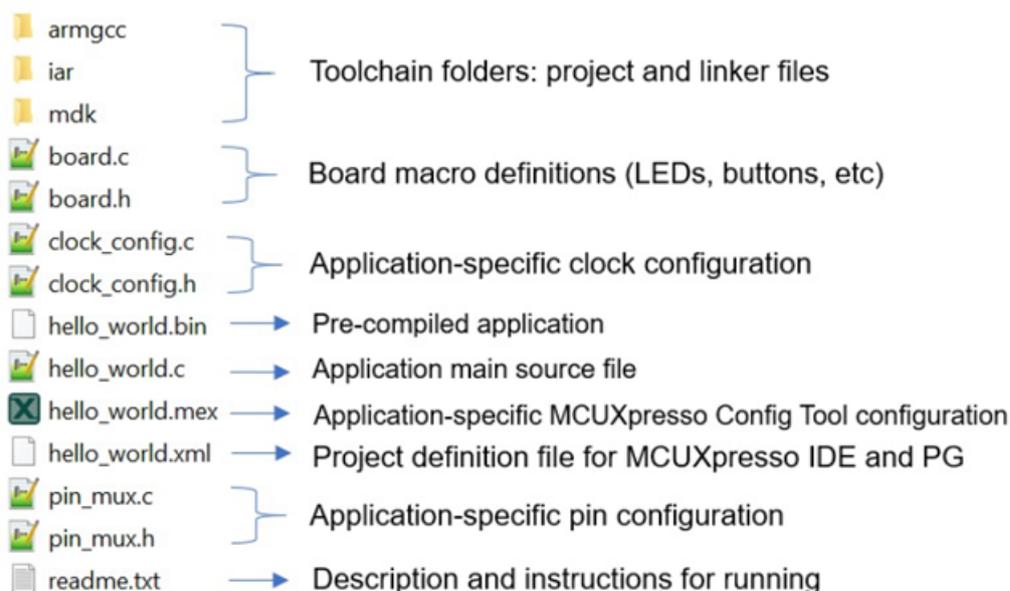
All files in the application folder are specific to that example, so it is easy to copy and paste an existing example to start developing a custom application based on a project provided in the MCUXpresso SDK.

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

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

- devices/<device_name>: The device's CMSIS header file, MCUXpresso SDK feature file, and a few other files

- devices/<device_name>/cmsis_drivers: All the CMSIS drivers for your specific MCU

- devices/<device_name>/drivers: All of the peripheral drivers for your specific MCU

- devices/<device_name>/<tool_name>: Toolchain-specific startup code, including vector table definitions

- devices/<device_name>/utilities: Items such as the debug console that are used by many of the example applications

- devices/<devices_name>/project: Project template used in CMSIS PACK new project creation

For examples containing middleware/stacks or an RTOS, there are references to the appropriate source code. Middleware source files are located in the middleware folder and 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.

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 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 the MCUXpresso SDK tree.

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 **OK** and wait until the import has finished.
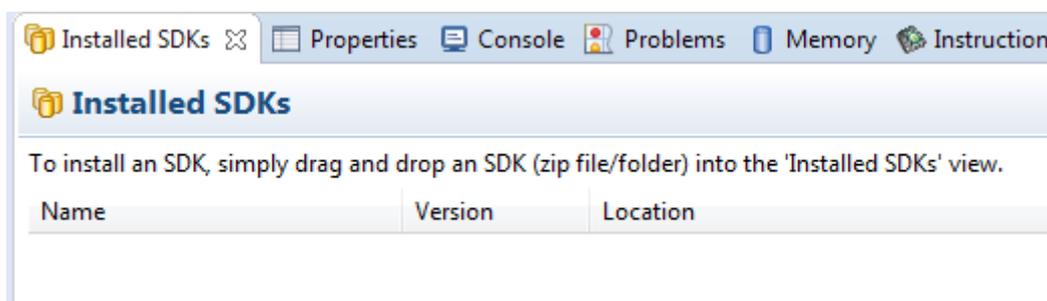
2. On the **Quickstart Panel**, click **Import SDK example(s)....**
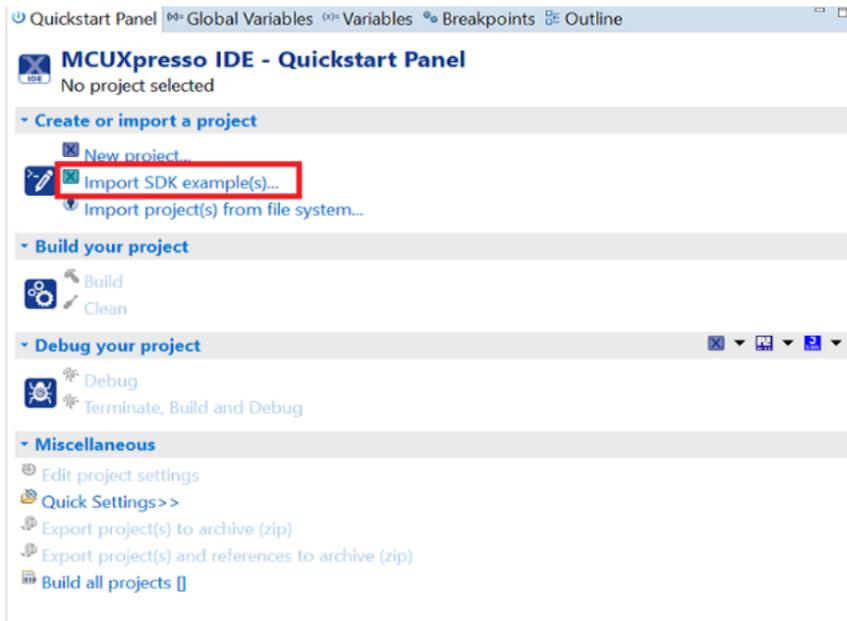
3. Expand the demo_apps folder and select hello_world.

4. Click **Next.**
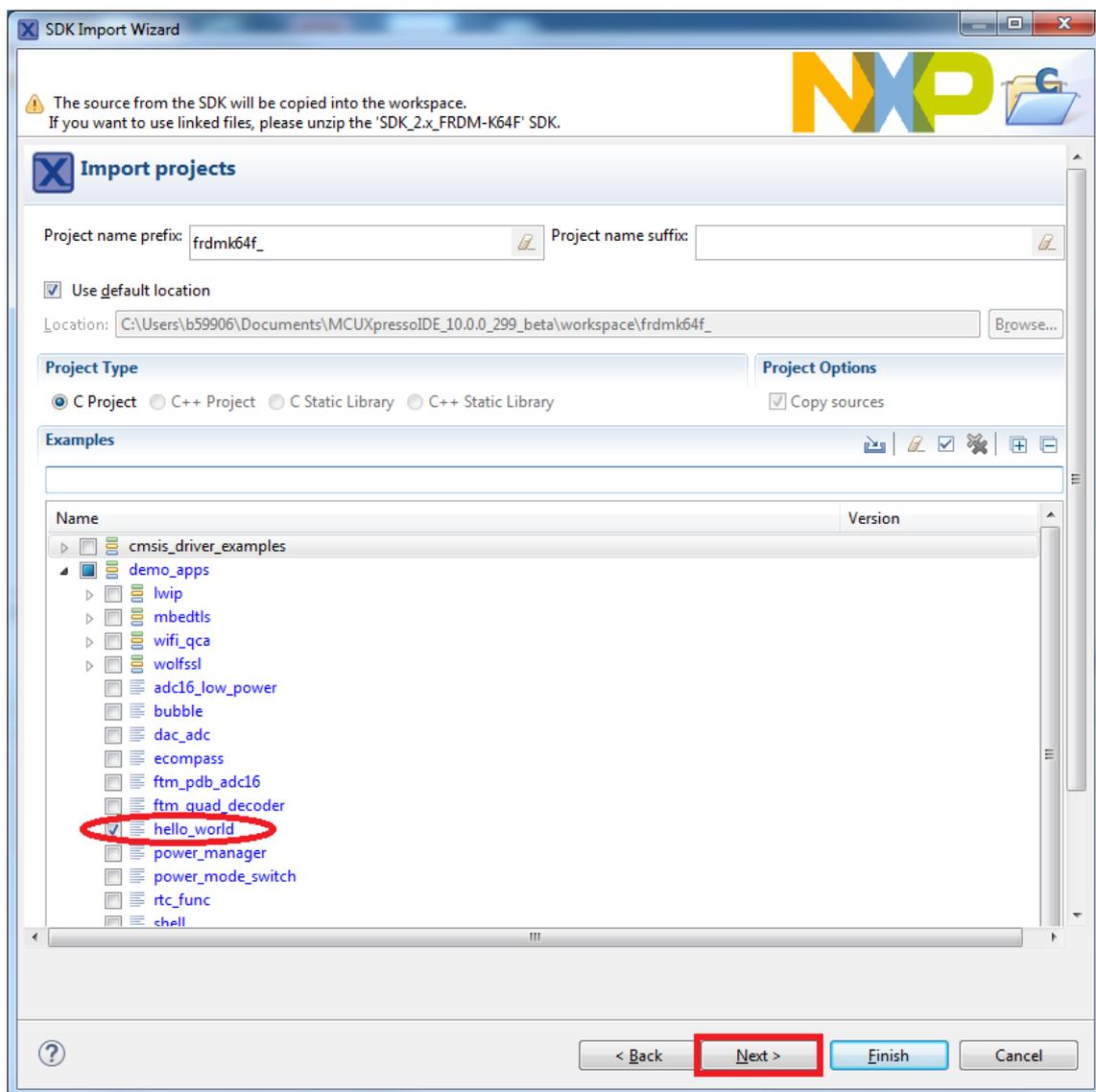
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminalfor demo applications such as adc_basic, adc_burst, adc_dma, and adc_interrupt. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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

To download and run the application, perform the following steps:

1. Ensure the host driver for the debugger firmware has been installed. See *On-board debugger*.

2. Connect the development platform to your PC via a USB cable.

3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see *How to determine COM port*. Configure the terminal with these settings:

   1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)

   2. No parity

3. 8 data bits
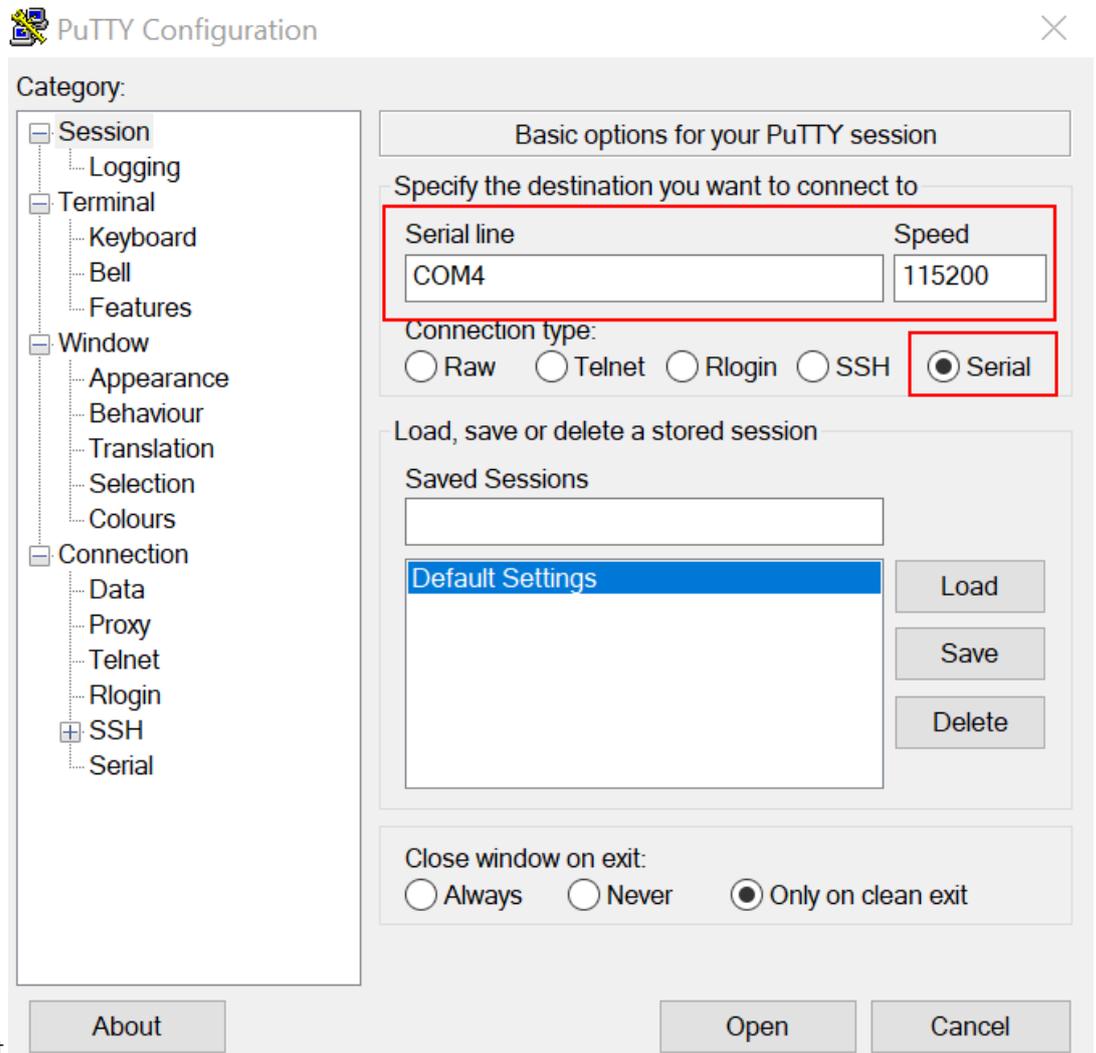
4. 1 stop bit

4. On the **Quickstart Panel**, click **Debug** to launch the debug session.

5. The first time you debug a project, the **Debug Emulator Selection** dialog is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)
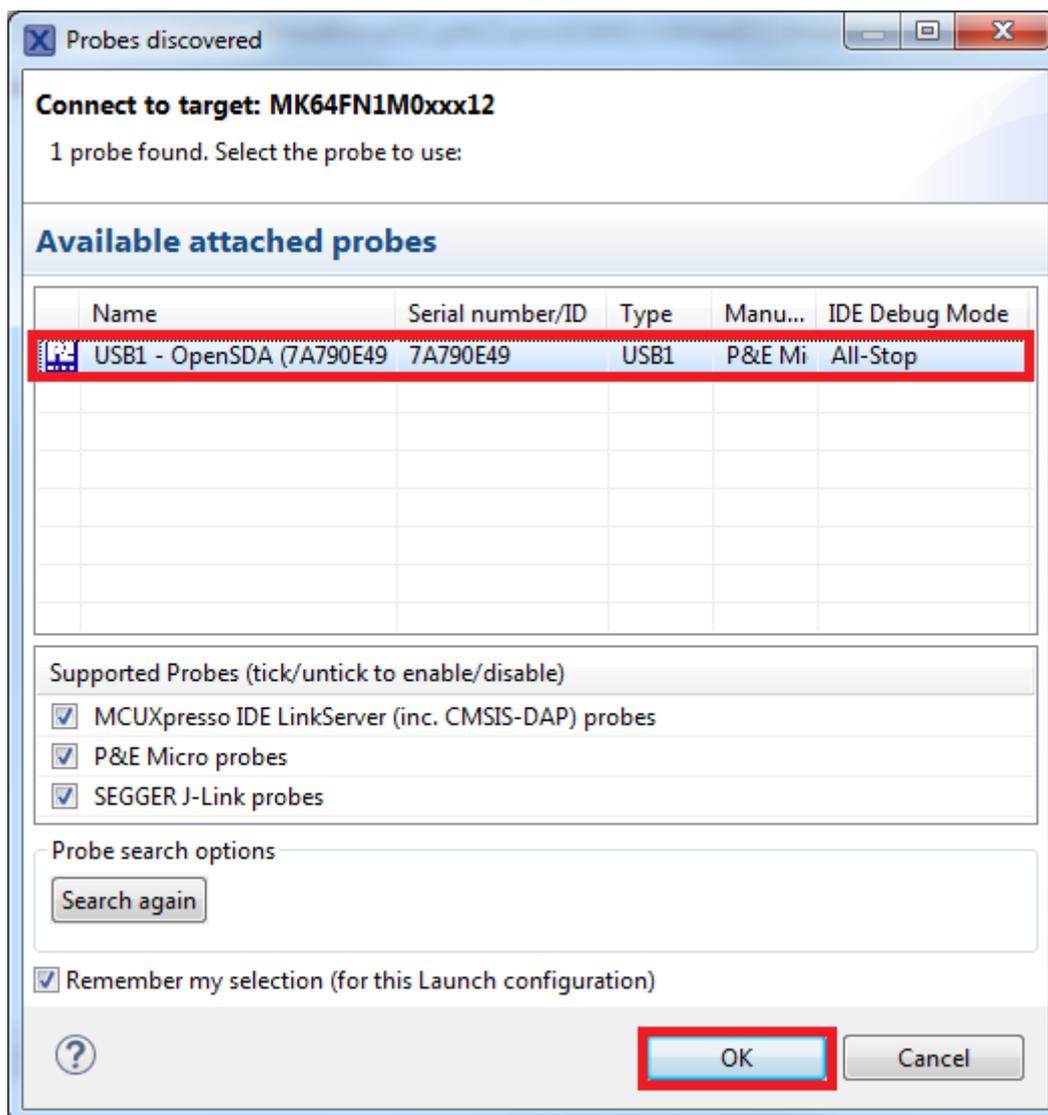
6. The application is downloaded to the target and automatically runs to main().

7. Start the application by clicking **Resume**.

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

Build a multicore example application   This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello_world example application targeted for the LPCXpresso54114 hardware platform is used as an example.

1. Multicore examples are imported into the workspace in a similar way as single core applications, explained in **Build an example application**. When the SDK zip package for LPCXpresso54114 is installed and available in the **Installed SDKs** view, click **Import SDK example(s)...** on the Quickstart Panel. In the window that appears, expand the **LPCxx** folder and select **LPC54114J256**. Then, select **lpcxpresso54114** and click **Next**.

2. Expand the multicore_examples/hello_world folder and select **cm4**. The cm0plus counterpart project is automatically imported with the cm4 project, because the multicore examples are linked together and there is no need to select it explicitly. Click **Finish**.
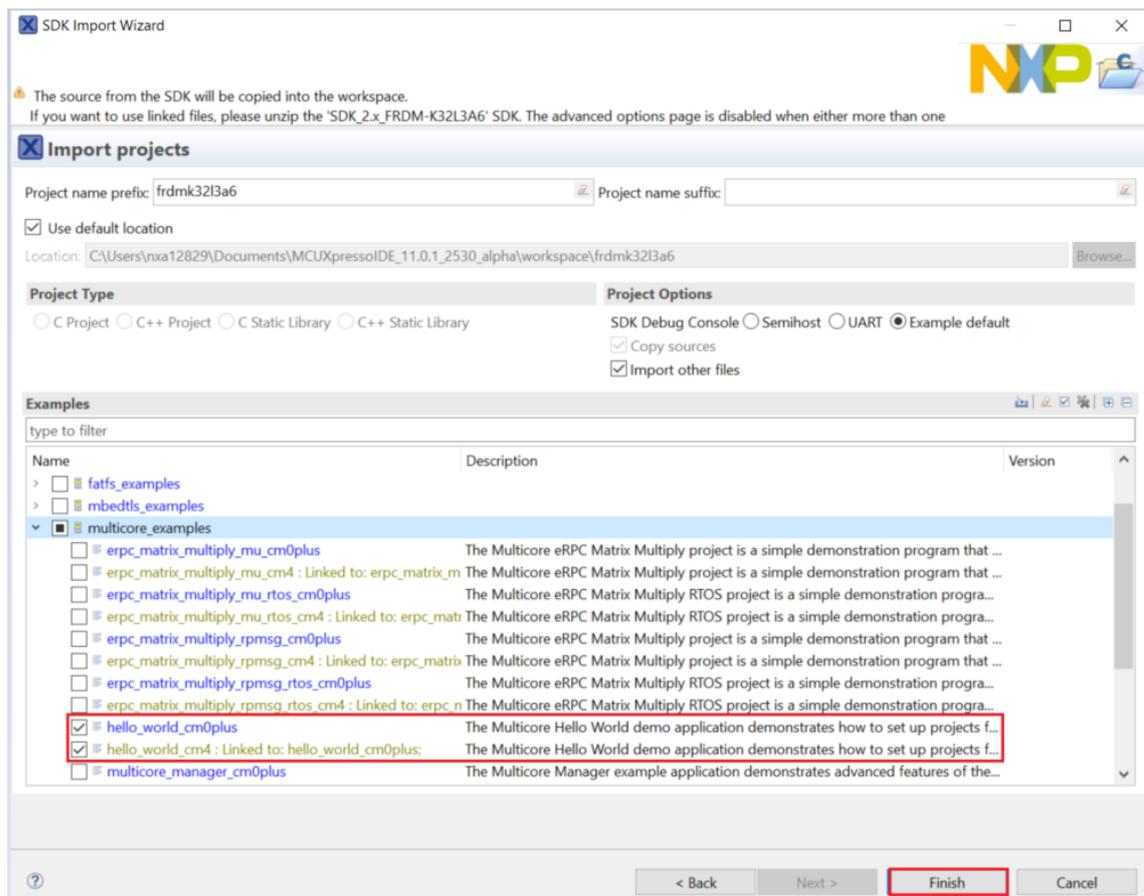
3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the lpcxpresso54114_multicore_examples_hello_world_cm4 project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in the figure. For this example, select **Debug**.
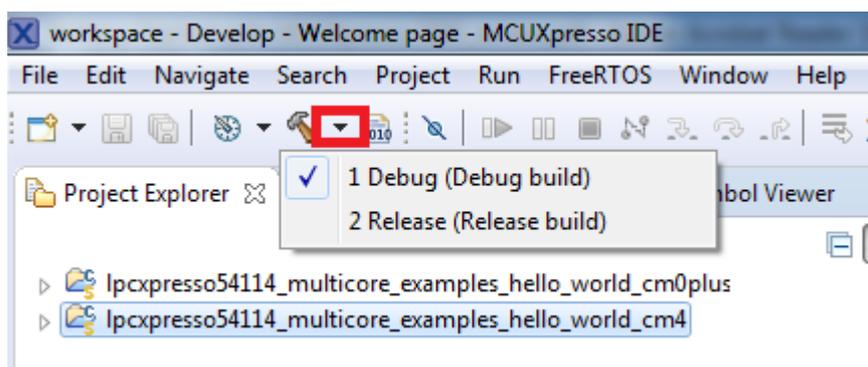
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

Note: When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **Build Configurations** -> **Set Active** -> **Release**. This alternate navigation using the menu item is **Project** -> **Build Configuration** -> **Set Active** -> **Release**. After switching to the **Release** build configuration, the build of the multicore example can be started by triggering the primary core application (cm4) build.
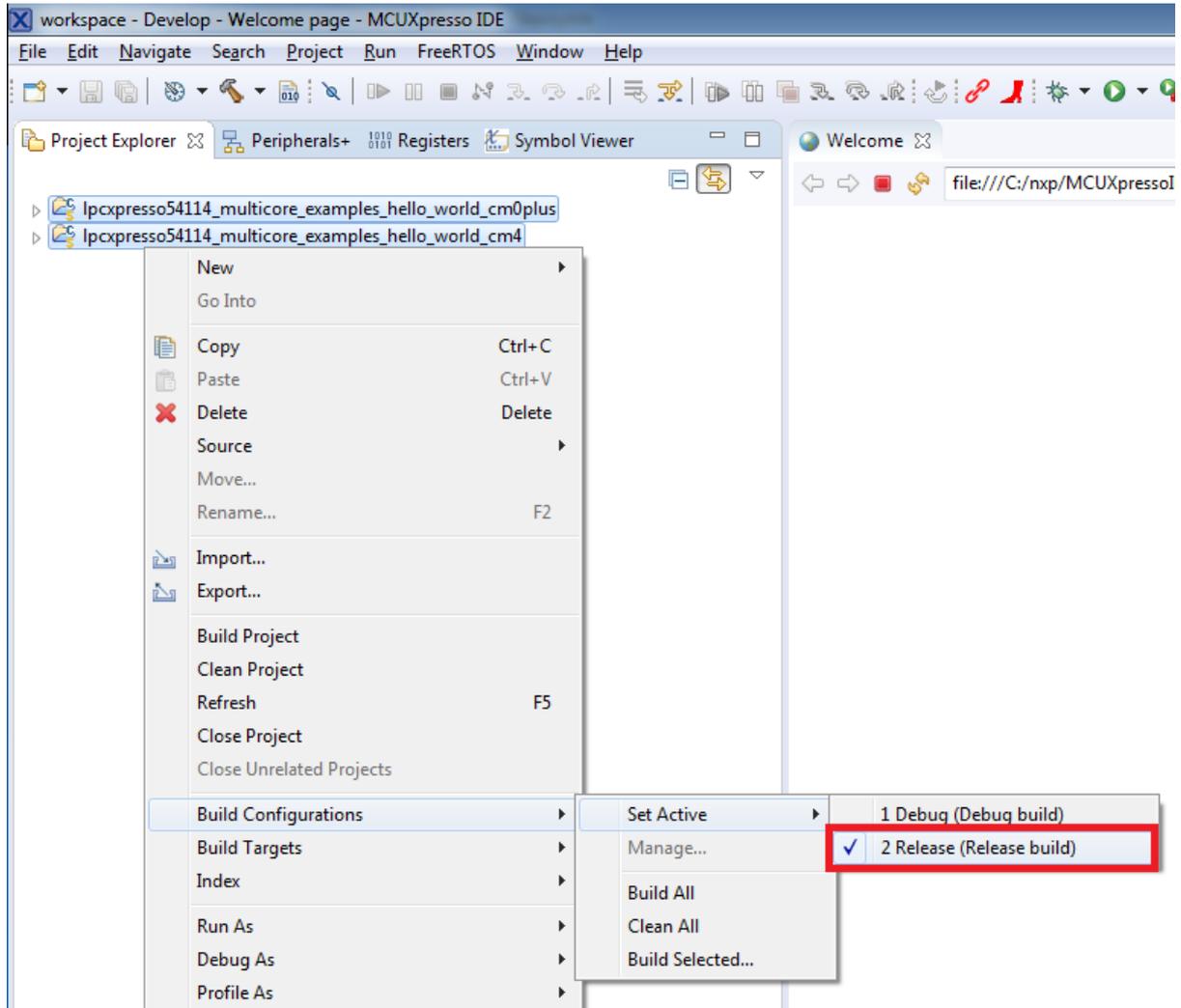
Run a multicore example application The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run an example application**. These steps are common for both single-core applications and the primary side of dual-core applications, ensuring both sides of the multicore application are properly loaded and started. However, there is one additional dialogue that is specific to multicore examples which requires selecting the target core. See the following figures as reference.
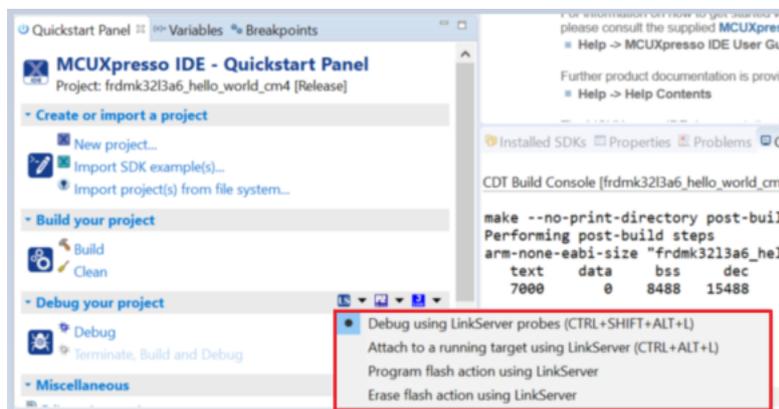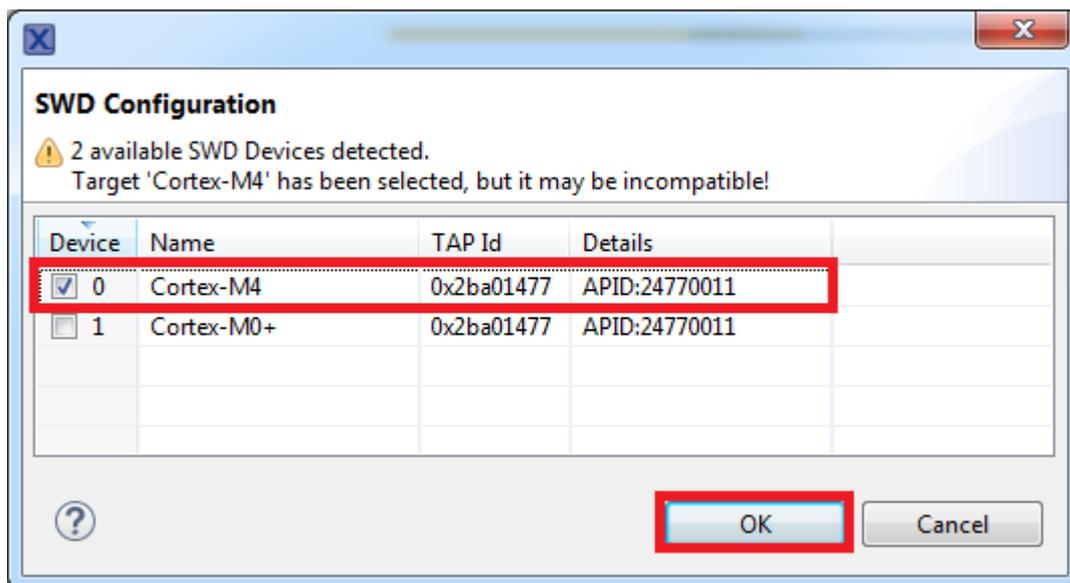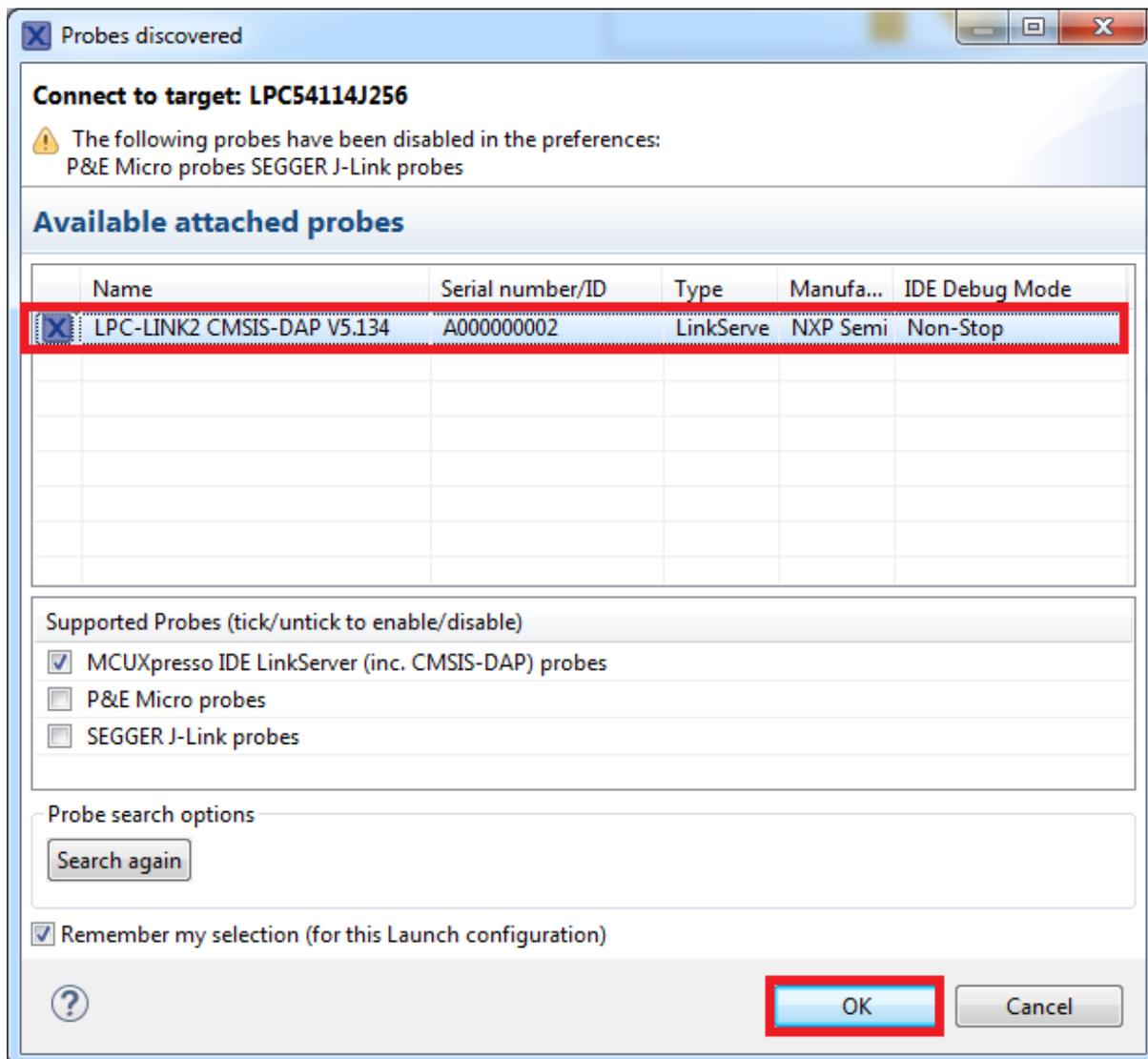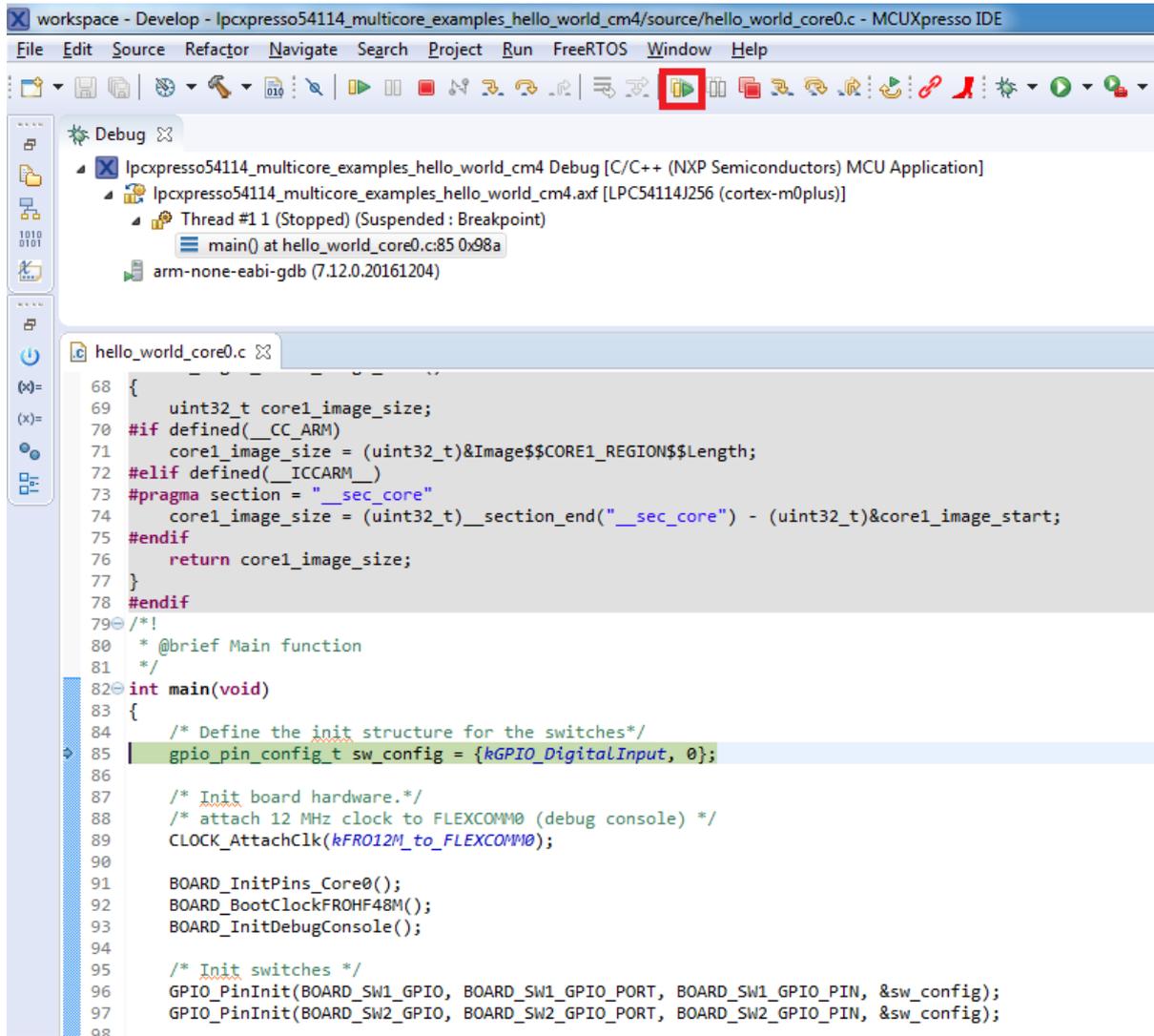
After clicking the "Resume All Debug sessions" button, the hello_world multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.
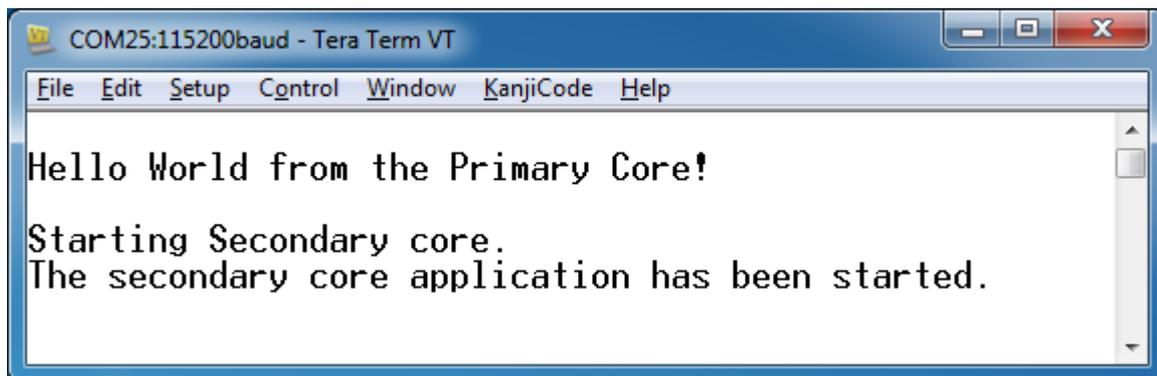
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps also for the auxiliary core application. Highlight the lpcxpresso54114_multicore_examples_hello_world_cm0plus project (multicore slave project) in the Project Explorer. On the Quickstart Panel, click "Debug 'lpcxpresso54114_multicore_examples_hello_world_cm0plus' [Debug]" to launch the second debug
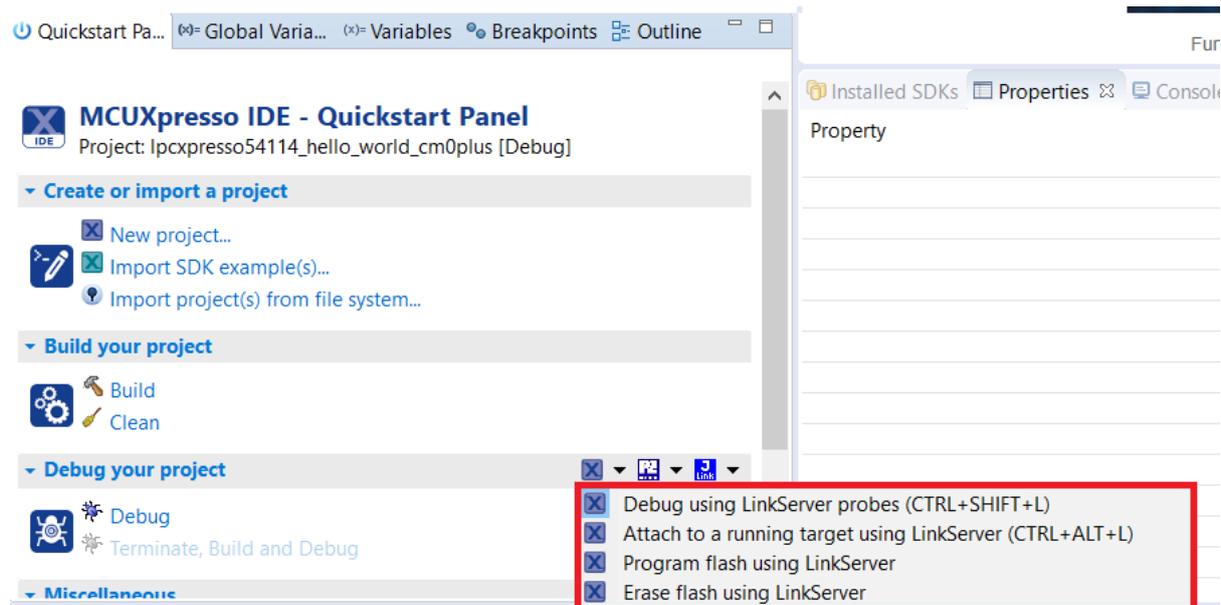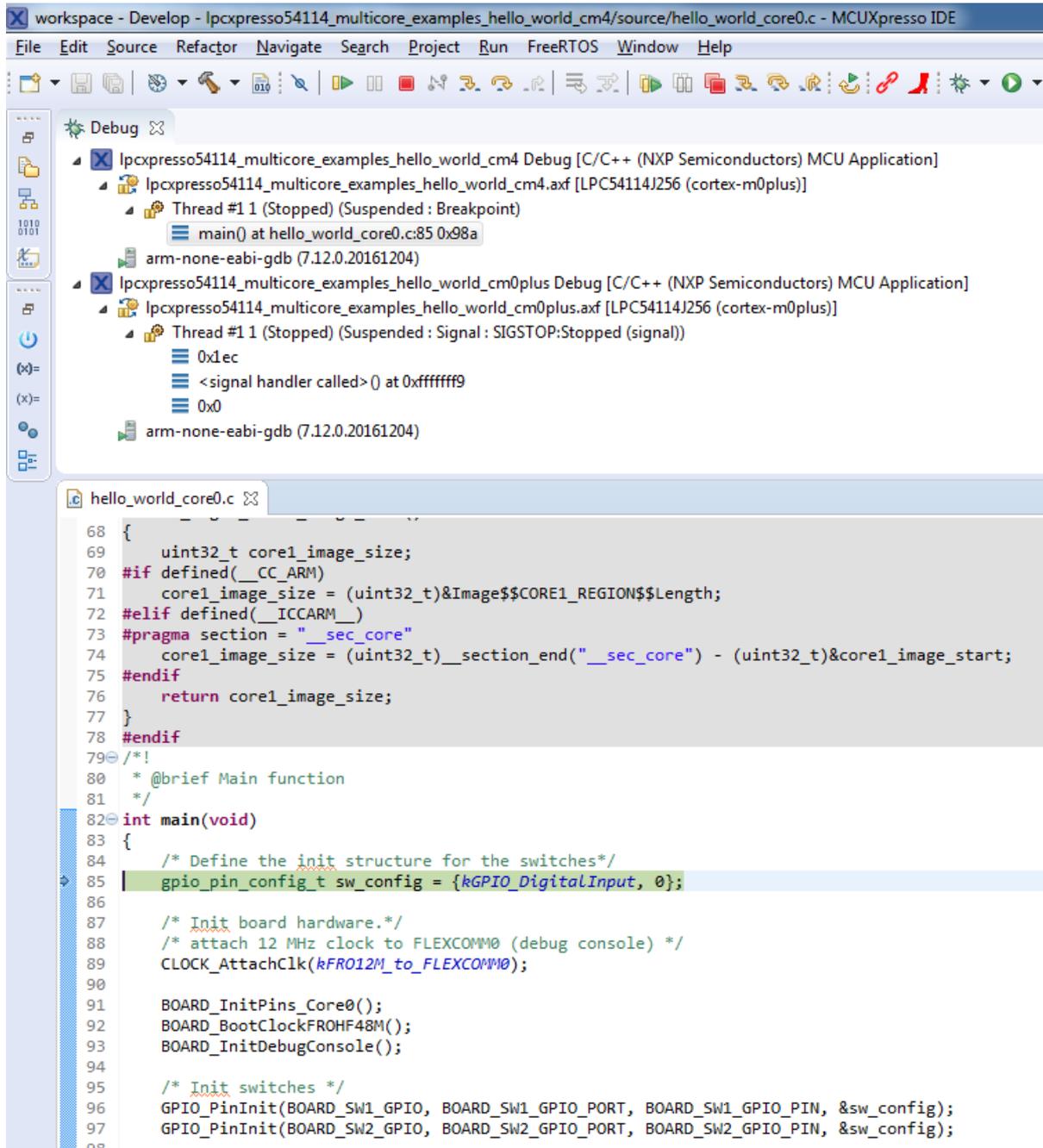
session.

Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the "Resume" button. The hello_world multicore application then starts running. The primary core application starts the auxiliary core application during runtime, and the auxiliary core application stops at the beginning of the main() function. The debug session of the auxiliary core application is highlighted. After clicking the "Resume" button, it is applied to the auxiliary core debug session. Therefore, the auxiliary core application continues its execution.
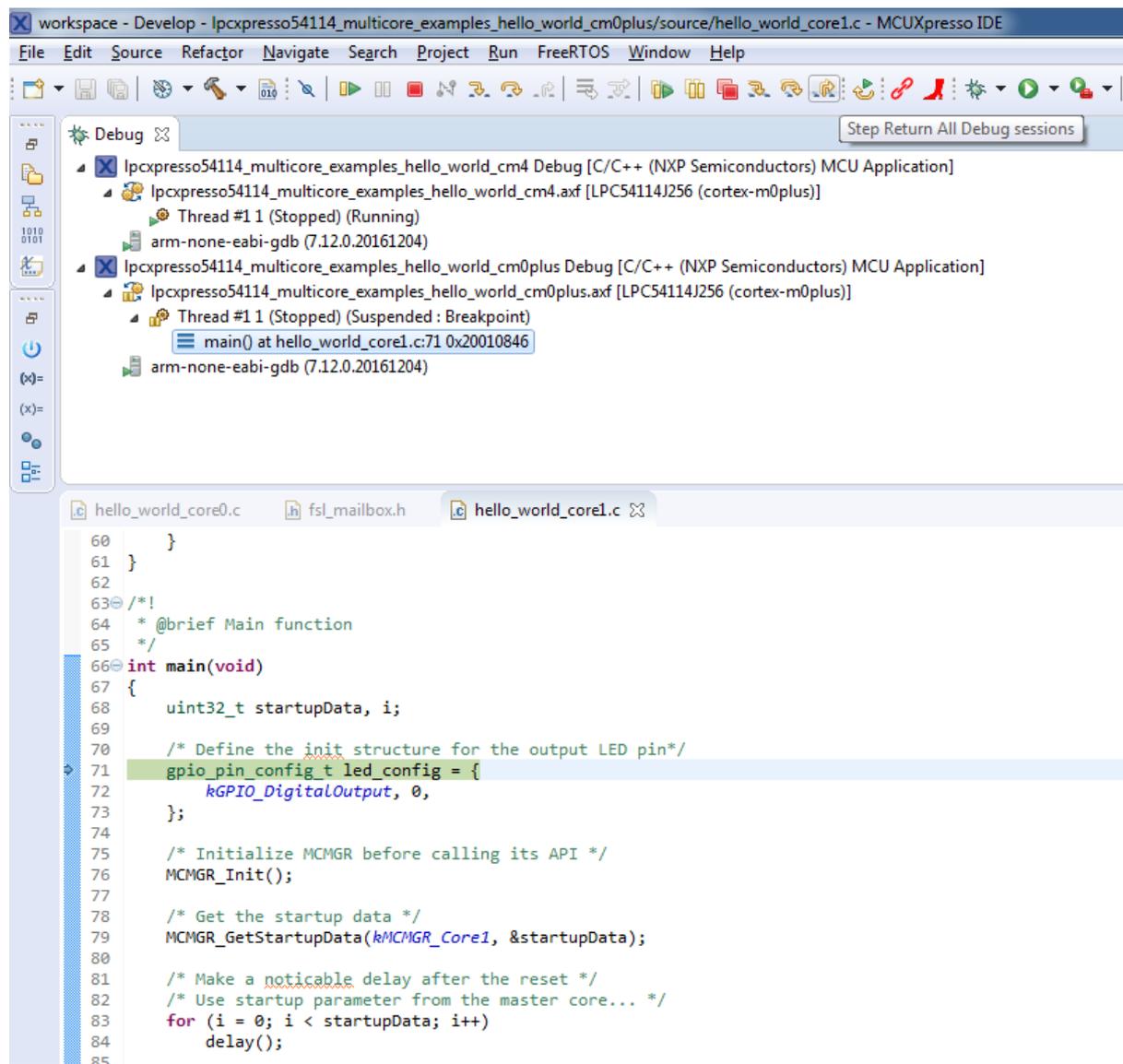
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the "Suspend" / "Resume" control button, or just using the "Suspend All Debug sessions" and the "Resume All Debug sessions" buttons.
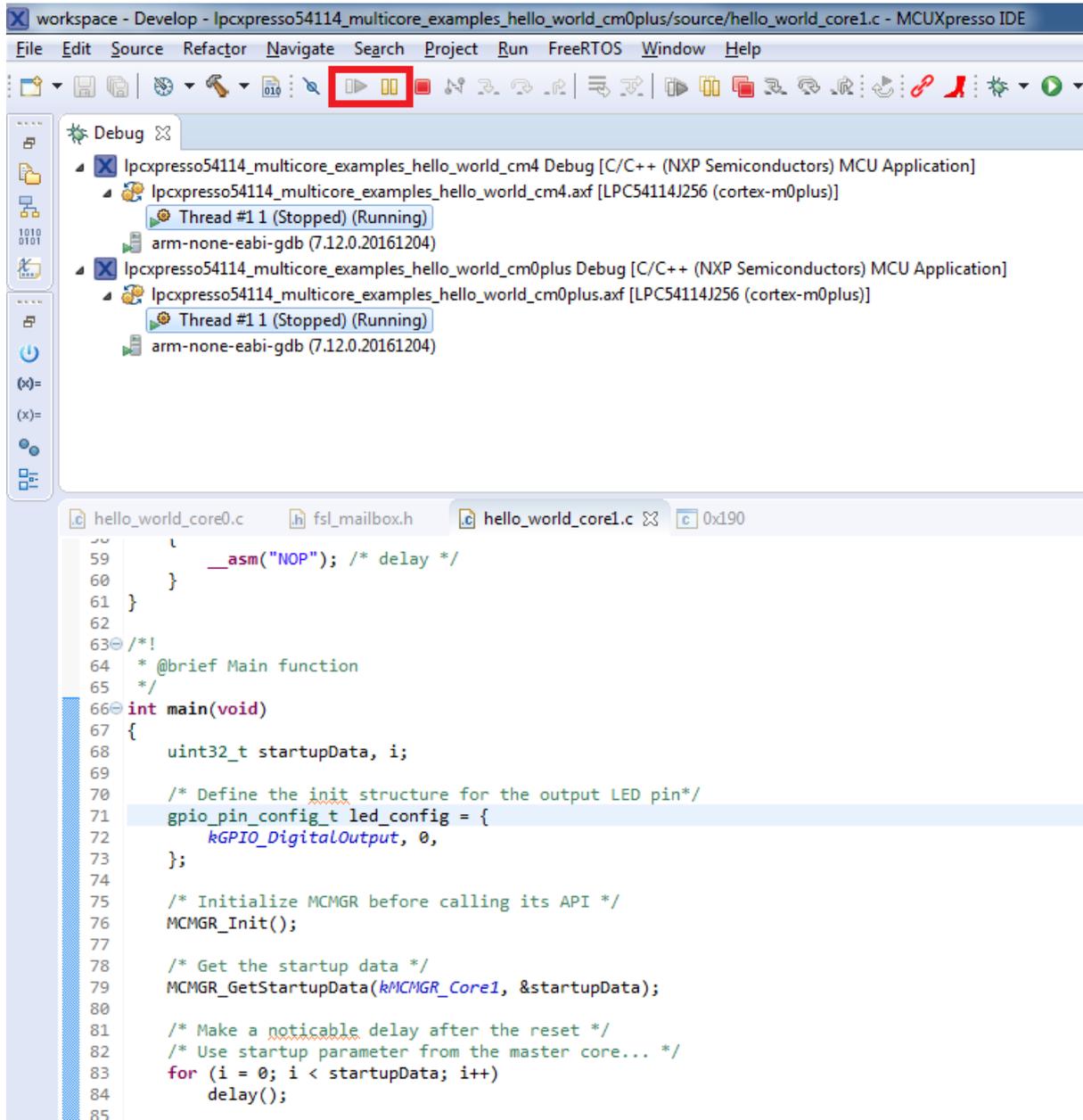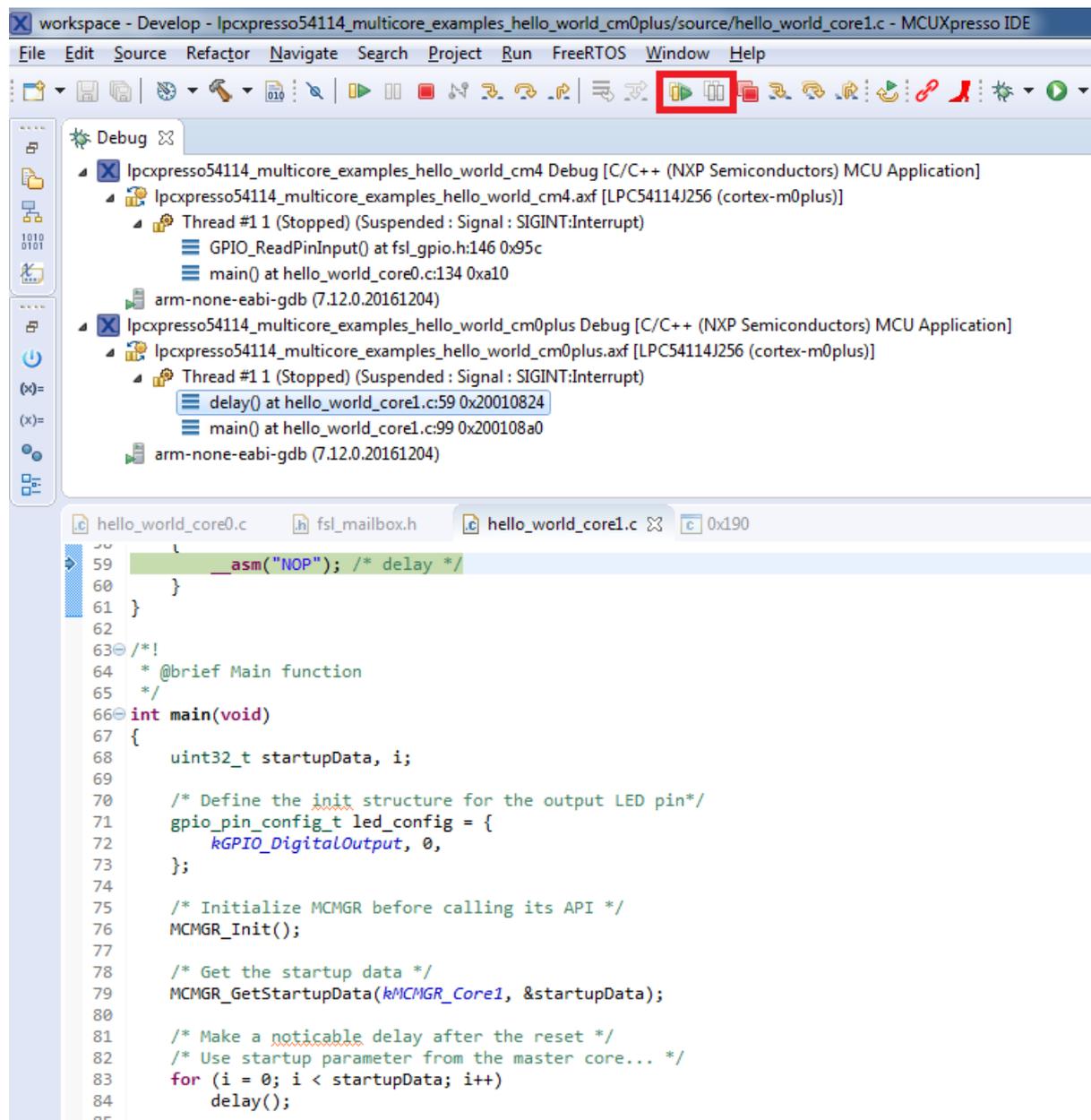
Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the hello_world example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though these steps can be applied to any TrustZone example application in the MCUXpresso SDK.

1. TrustZone examples are imported into the workspace in a similar way as single core applications. When the SDK zip package for MIMXRT595-EVK is installed and available in the **Installed SDKs** view, click **Import SDK example(s)...** on the Quickstart Panel. In the window that appears, expand the **MIMXRT500** folder and select **MIMXRT595S**. Then, select **evkmimxrt595** and click **Next**.

2. Expand the trustzone_examples/ folder and select hello_world_s. Because TrustZone examples are linked together, the non-secure project is automatically imported with the secure project, and there is no need to select it explicitly. Then, click **Finish**.
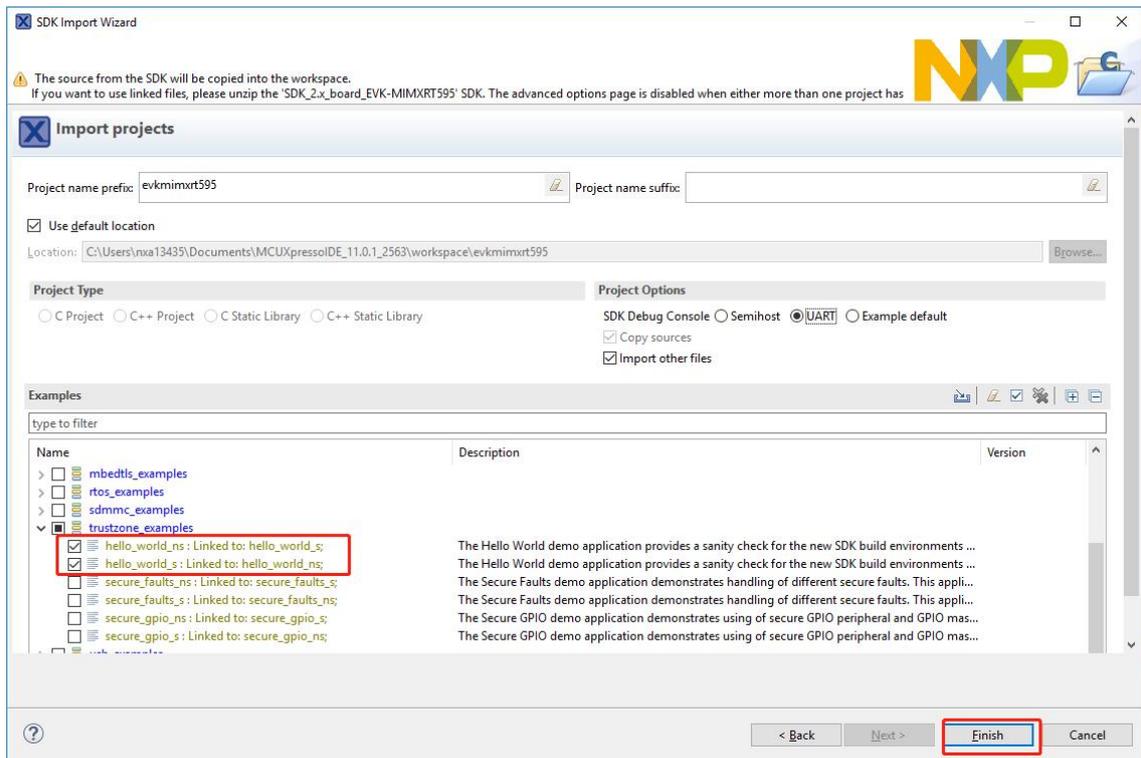
3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the evkmimxrt595_hello_world_s project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in following figure. For this example, select the **Debug** target.
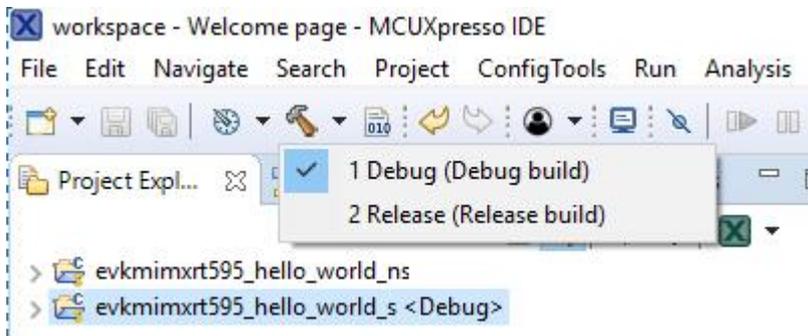

The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when the secure project since CMSE library is not ready.

Note: When the **Release** build is requested, it is necessary to change the build configuration of both the secure and non-secure application projects first. To do this, select both projects in the Project Explorer view by clicking to select the first project, then using shift-click or control-click to select the second project. Right click in the Project Explorer view to display the context-sensitive menu and select **Build Configurations** > **Set Active** >**Release**. This is also possible by using the menu item of **Project** > **Build Configuration** >**Set Active** >**Release**. After switching to the **Release** build configuration. Build the application for the secure project first.
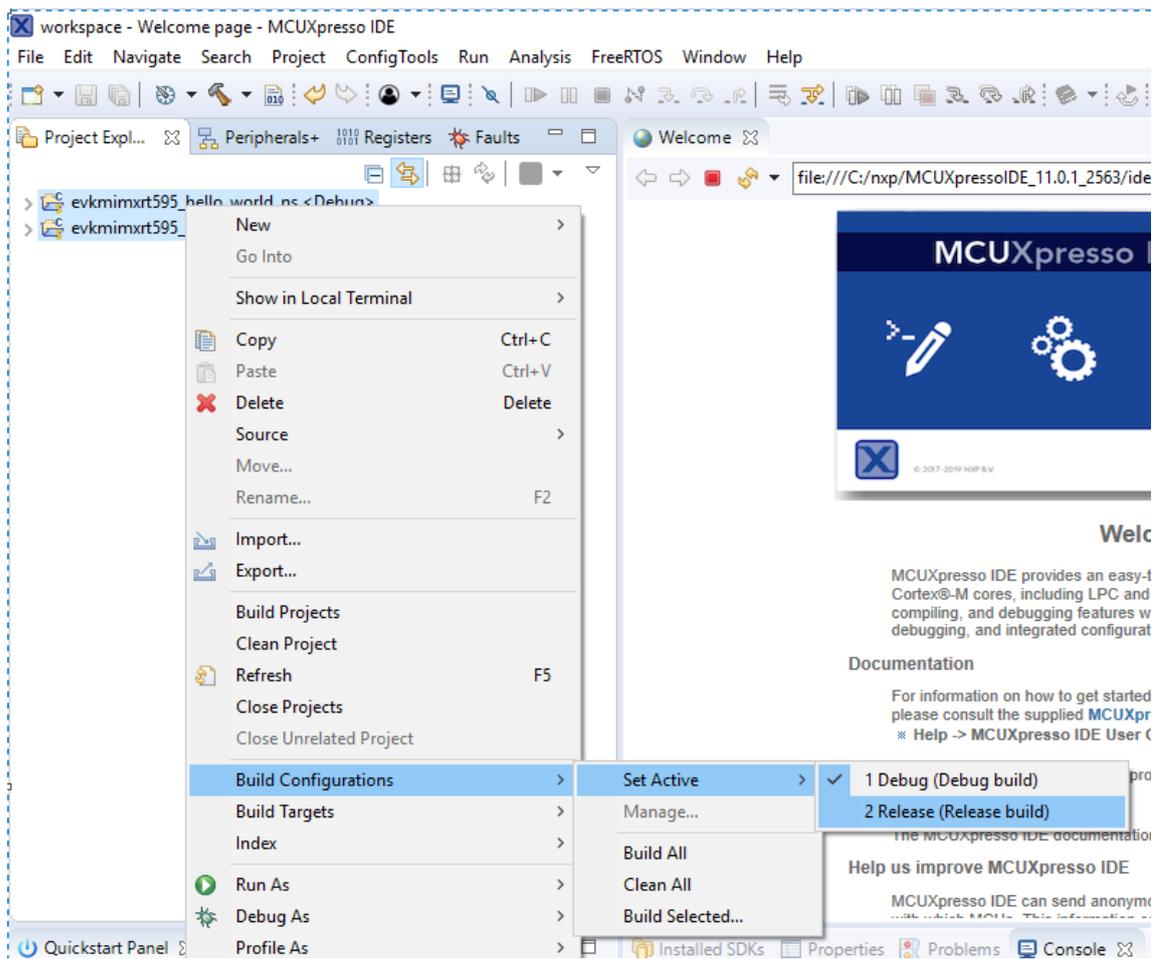
Run a TrustZone example application To download and run the application, perform all steps as described in **Run an example application**. These steps are common for single core, and TrustZone applications, ensuring $<$board_name$>$_hello_world_s is selected for debugging.

In the Quickstart Panel, click **Debug** to launch the second debug session.
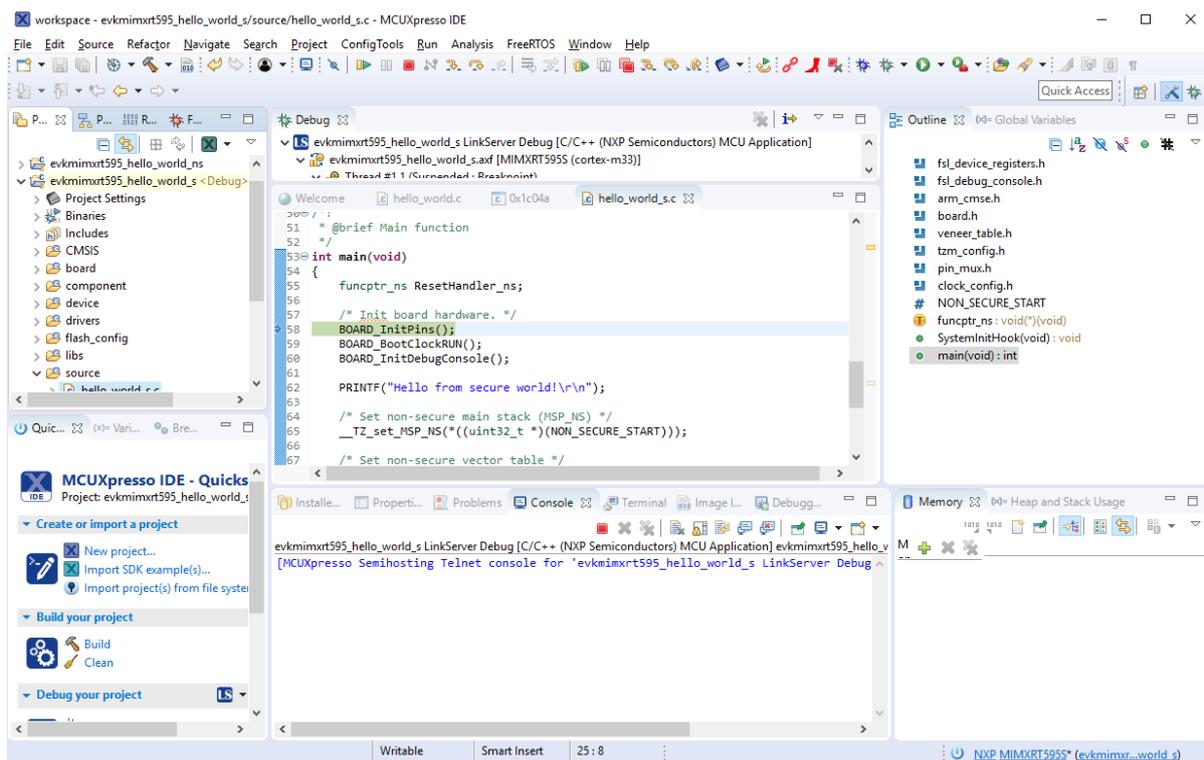
Now, the TrustZone sessions should be opened. Click **Resume**. The hello_world TrustZone application then starts running, and the secure application starts the non-secure application during runtime.

Run a demo application using IAR   This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

Note: IAR Embedded Workbench for Arm version 8.32.3 is used in the following example, and the IAR toolchain should correspond to the latest supported version, as described in the *MCUX-presso SDK Release Notes*.

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

   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**.
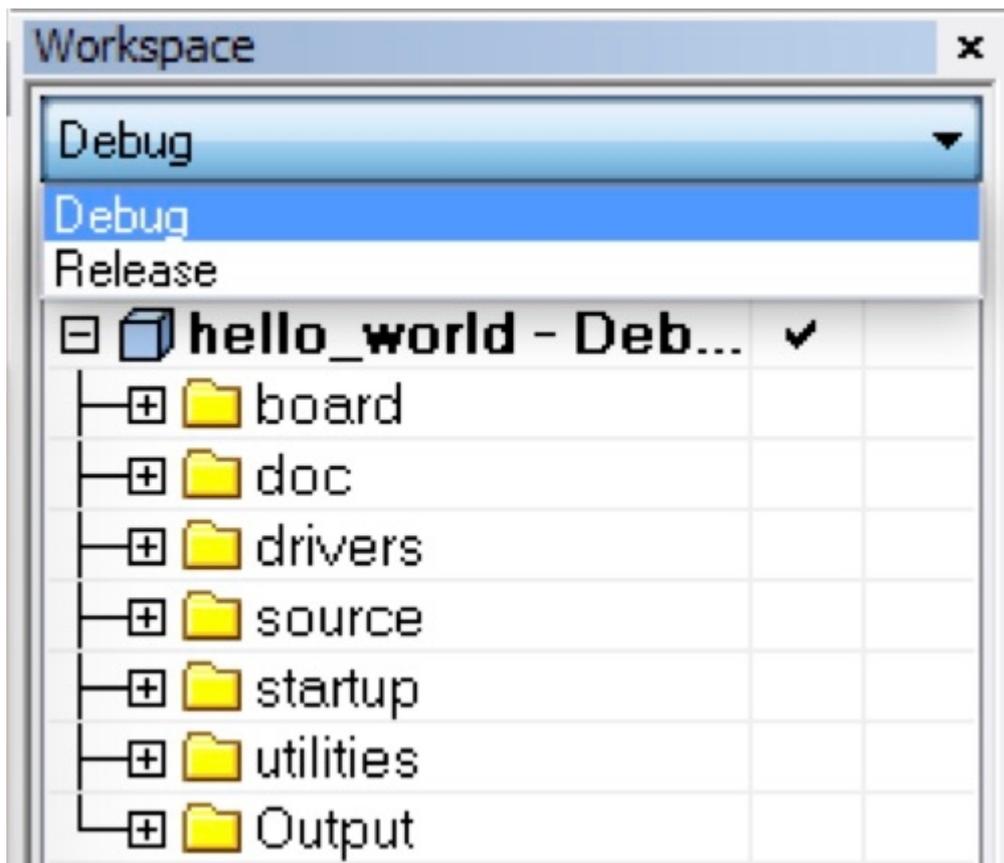
3. To build the demo application, click **Make**, highlighted in red in following figure.
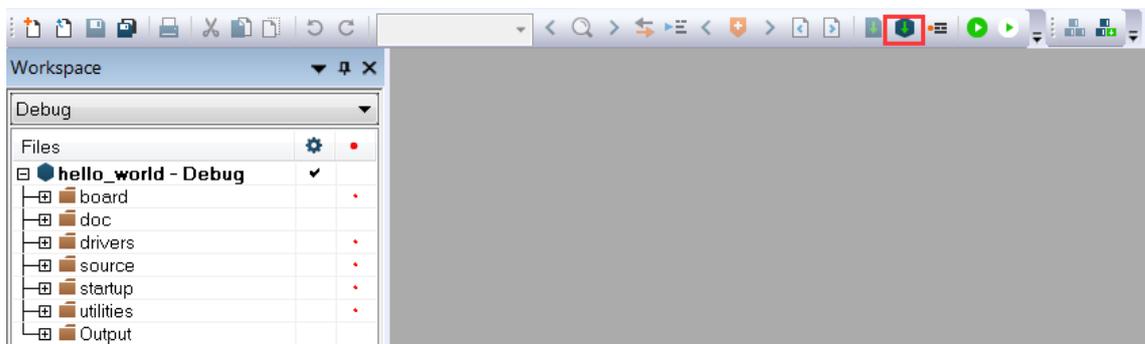
4. The build completes without errors.

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

1. Ensure the host driver for the debugger firmware has been installed. See *On-board debugger*.

2. Connect the development platform to your PC via USB cable.

3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see *How to determine COM port*). Configure the terminal with these settings:

    1. 115200   or   9600   baud   rate,   depending   on   your   board   (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)

    2. No parity

    3. 8 data bits
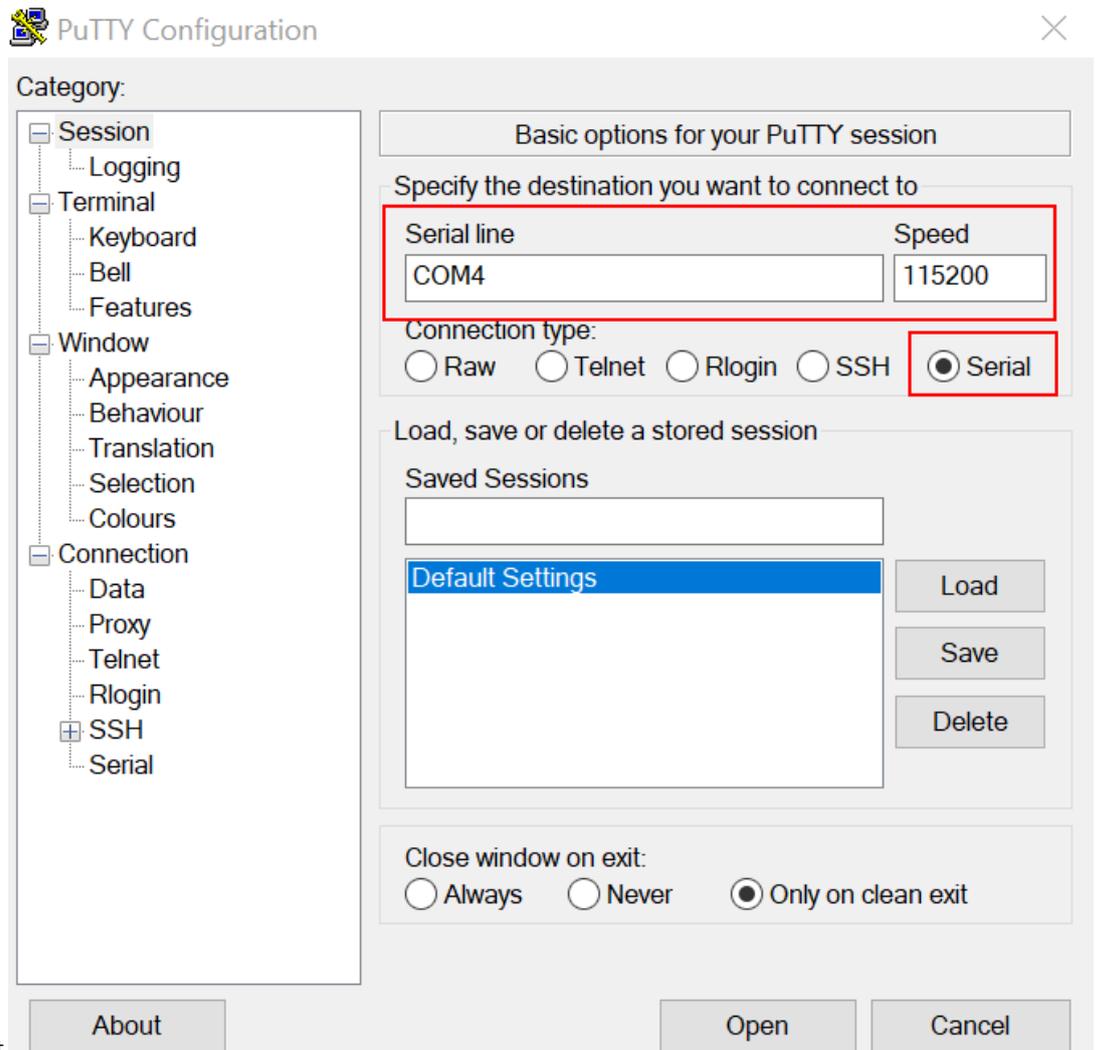
4. 1 stop bit

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

5. The application is then downloaded to the target and automatically runs to the $main()$ function.
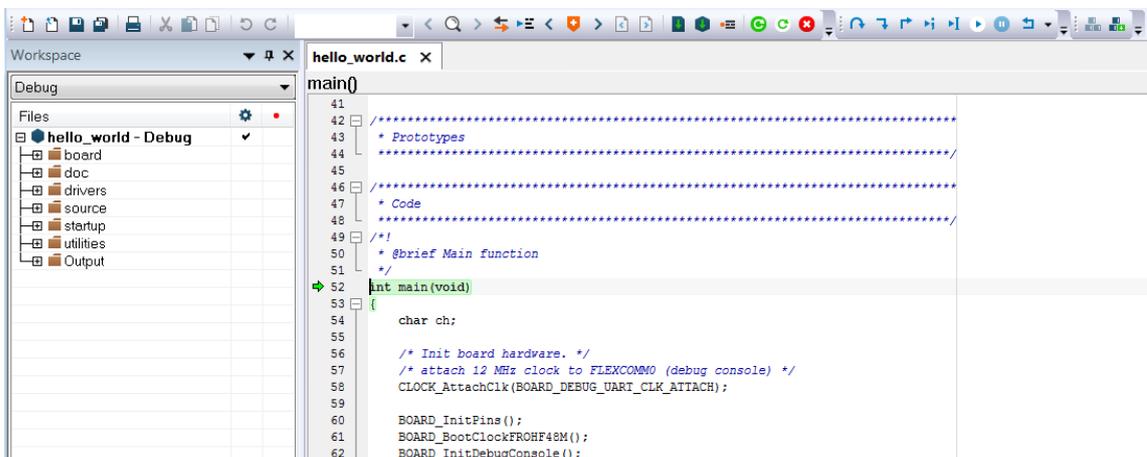
6. Run the code by clicking the **Go** button.

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


Build a multicore example application   This section describes the steps to build and run a dual-core application. The demo applications workspace files are located in this folder:

<install_dir>/boards/<board_name>/multicore_examples/<application_name>/<core_type>/iar

Begin with a simple dual-core version of the Hello World application. The multicore Hello World IAR workspaces are located in this folder:

<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/iar/hello_world_cm0plus.
↪eww

<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/iar/hello_world_cm4.eww

Build both applications separately by clicking the **Make** button. Build the application for the auxiliary core (cm0plus) first, because the primary core application project (cm4) must know the auxiliary core application binary when running the linker. It is not possible to finish the primary core linker when the auxiliary core application binary is not ready.

Run a multicore example application   The primary core debugger handles flashing both primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 4 as described in **Run an example application**. These steps are common for both single core and dual-core applications in IAR.

After clicking the "Download and Debug" button, the auxiliary core project is opened in the separate EWARM instance. Both the primary and auxiliary images are loaded into the device flash memory and the primary core application is executed. It stops at the default C language entry point in the *main()*function.

Run both cores by clicking the "Start all cores" button to start the multicore application.

During the primary core code execution, the auxiliary core is released from the reset. The hello_world multicore application is now running and a banner is displayed on the terminal. If this does not appear, check the terminal settings and connections.
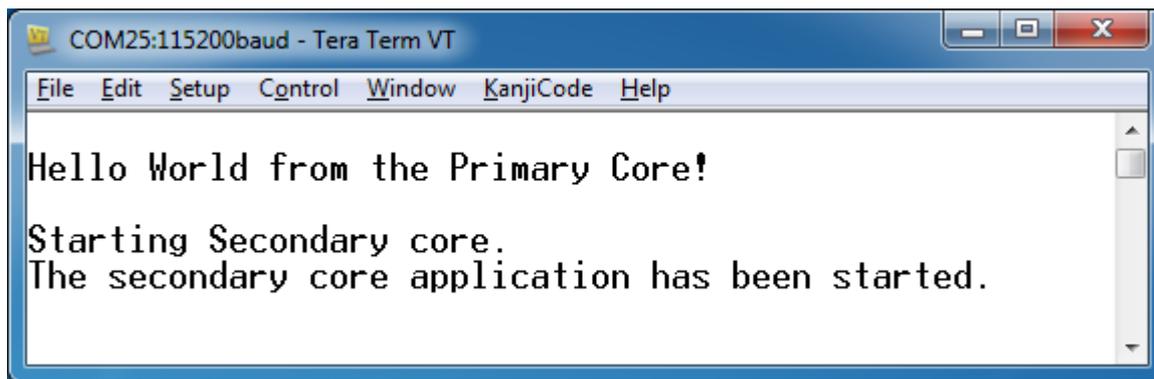
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and is running correctly. When both cores are running, use the "Stop all cores", and "Start all cores" control buttons to stop or run both cores simultaneously.
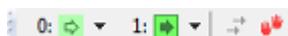
Build a TrustZone example application   This section describes the particular steps that must be done in order to build and run a TrustZone application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/iar/
↪<application_name>__ns/iar
```

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/iar/
↪<application_name>__s/iar
```

Begin with a simple TrustZone version of the Hello World application. The TrustZone Hello World IAR workspaces are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_
↪ns.eww
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/iar/hello_world_s.
↪eww
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/iar/hello_world.eww
```

This project hello_world.eww contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because the non-secure project must know the secure project, since the CMSE library is running the linker. It is not possible to finish the non-secure project linker with the secure project since CMSE library is not ready.

Run a TrustZone example application   The secure project is configured to download both secure and non-secure output files, so debugging can be fully managed from the secure project. To download and run the TrustZone application, switch to the secure application project and perform steps 1 – 4 as described in **Run an example application**. These steps are common for both single core, and TrustZone applications in IAR. After clicking **Download and Debug**, both the secure and non-secure images are loaded into the device memory, and the secure application is executed. It stops at the Reset_Handler function.
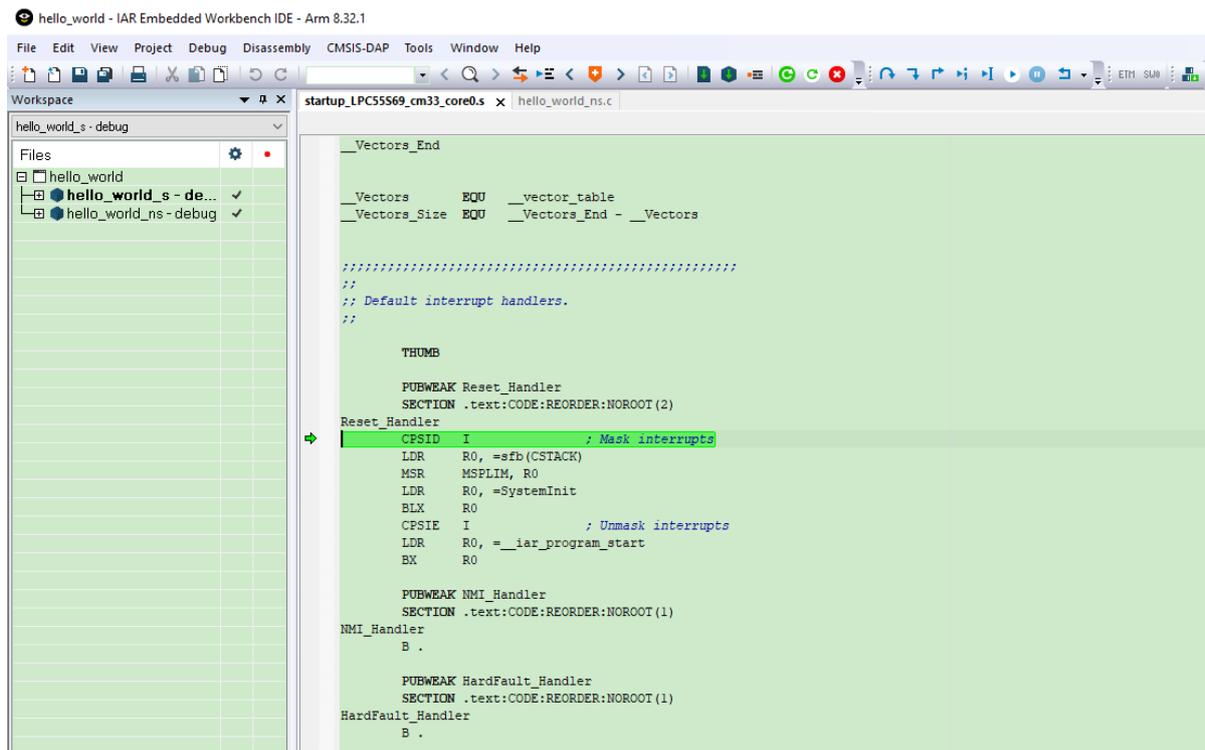
Run the code by clicking **Go** to start the application.

The TrustZone hello_world application is now running and a banner is displayed on the terminal. If this is not true, check your terminal settings and connections.
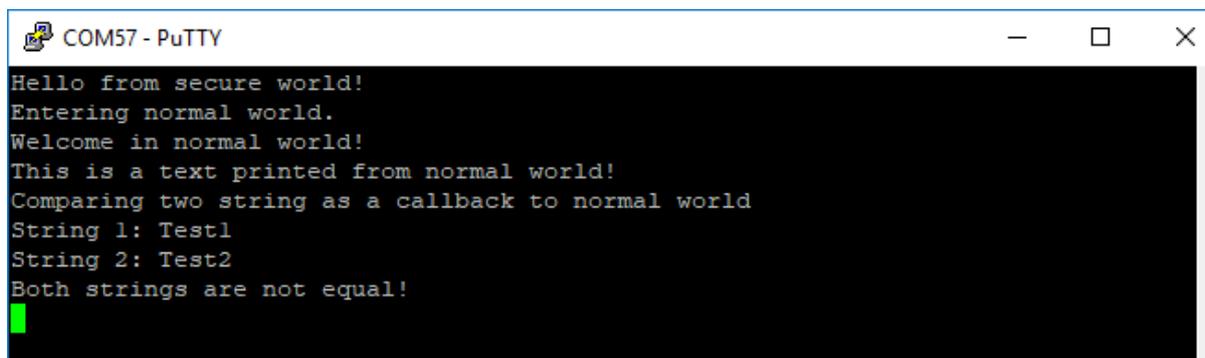
Note: If the application is running in RAM (debug/release build target), in **Options**>**Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the _ns download issue on i.MXRT500.
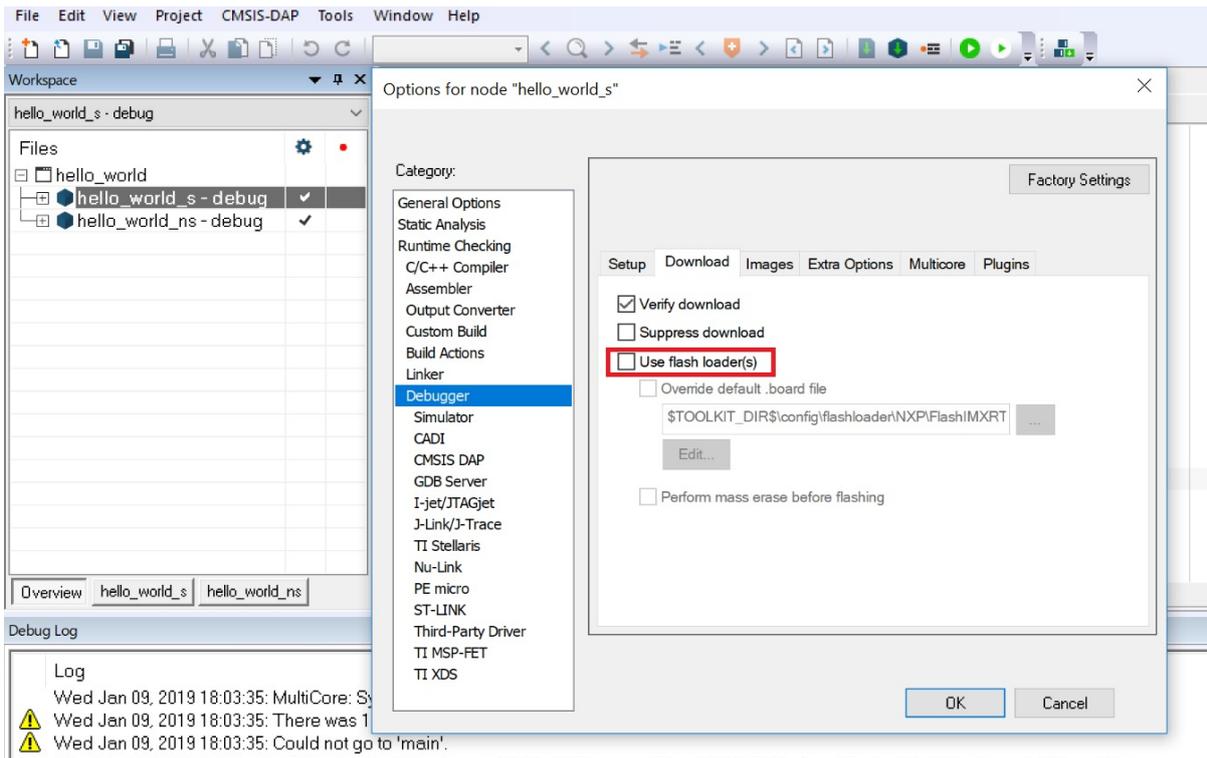
Run a demo using Keil MDK/μVision This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

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

1. Open the MDK IDE, which is called μVision. In the IDE, select the **Pack Installer** icon.
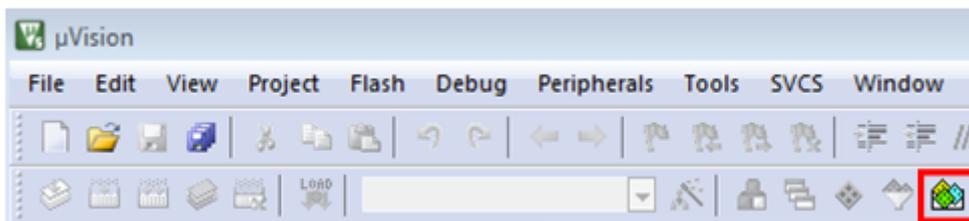
2. After the installation finishes, close the Pack Installer window and return to the μVision IDE.

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:

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

3. The build completes without errors.

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

1. Ensure the host driver for the debugger firmware has been installed. See *On-board debugger*.

2. Connect the development platform to your PC via USB cable using USB connector.

3. Open the terminal application on the PC, such as PuTTY or TeraTerm and connect to the debug serial port number (to determine the COM port number, see *How to determine COM port*. Configure the terminal with these settings:

   1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)

   2. No parity

   3. 8 data bits
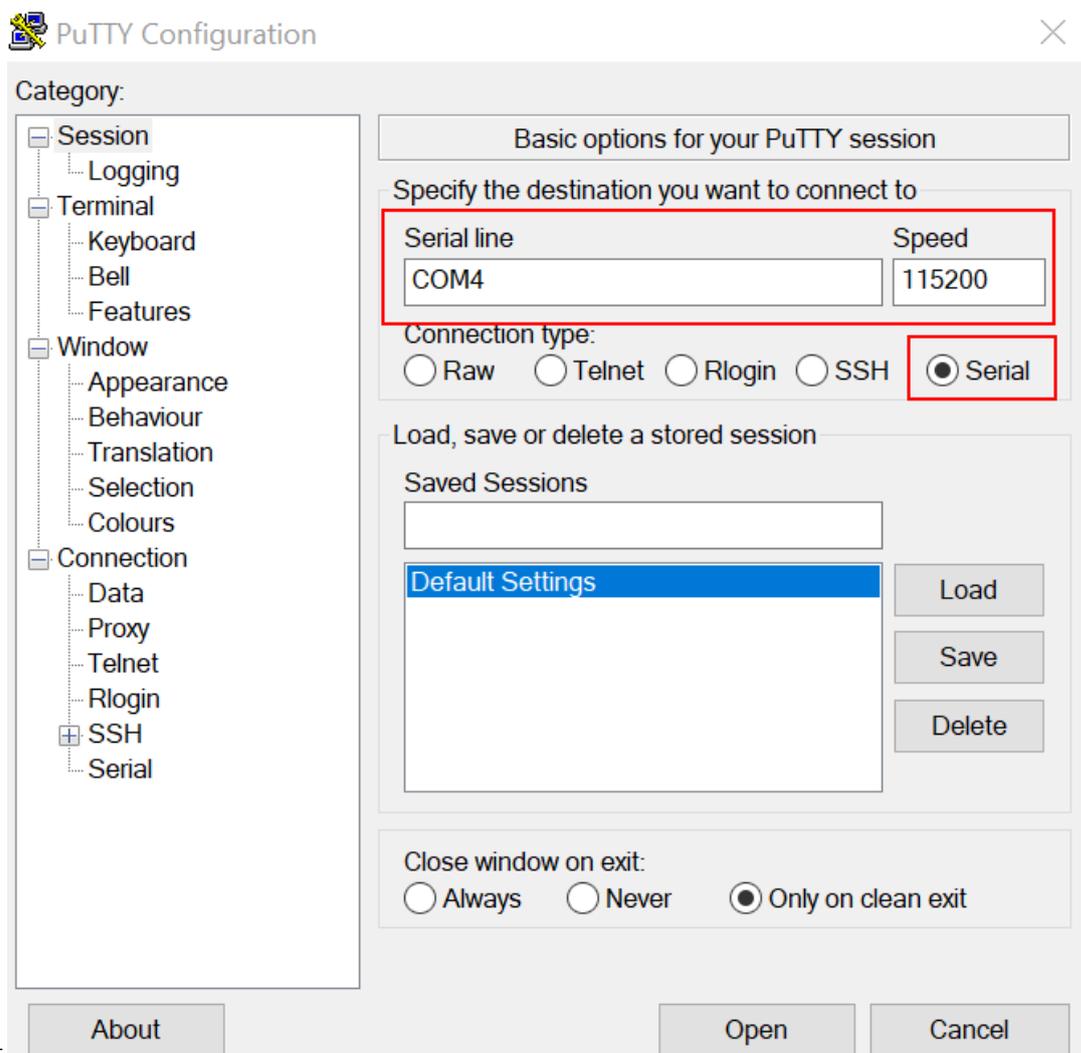
   4. 1 stop bit

4. In µVision, after the application is built, click the **Download** button to download the application to the target.

5. After clicking the **Download** button, the application downloads to the target and is running. To debug the application, click the **Start/Stop Debug Session** button, highlighted in red.
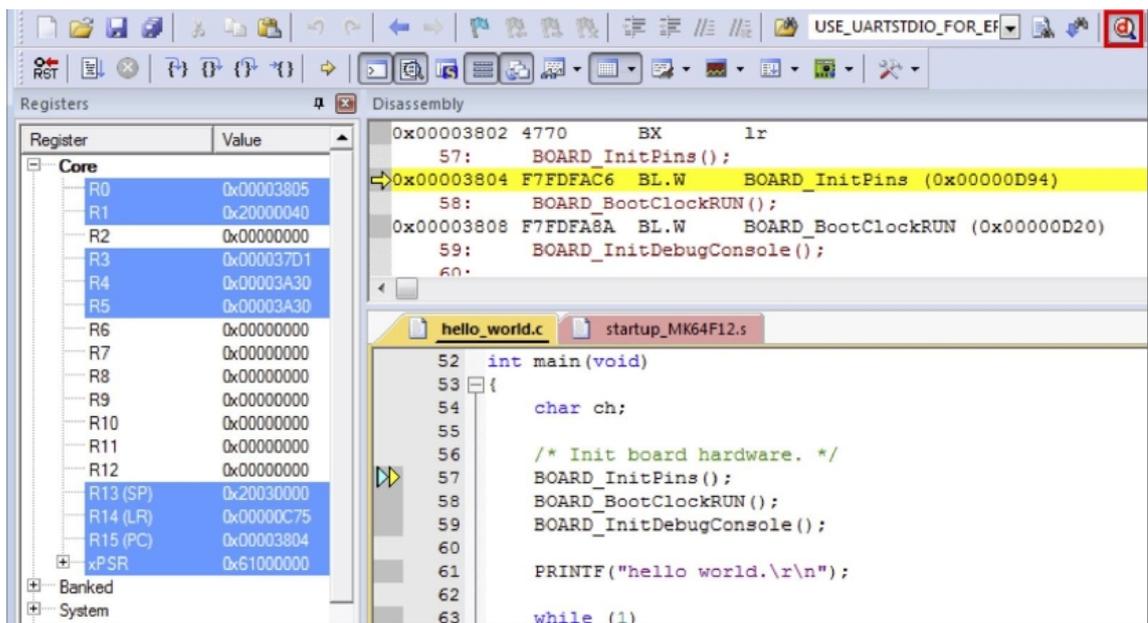
6. Run the code by clicking the **Run** button to start the application.
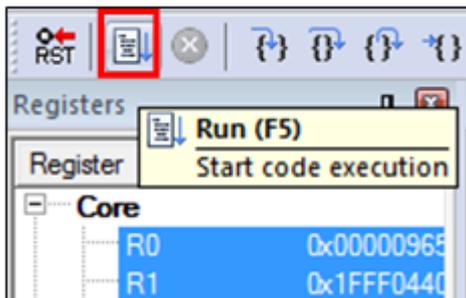
The hello_world application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.

Build a multicore example application   This section describes the steps to build and run a dual-core application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/multicore_examples/<application_name>/<core_type>/mdk
```

Begin with a simple dual-core version of the Hello World application. The multicore Hello World Keil MSDK/µVision workspaces are located in this folder:

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
↪cm0plus.uvmpw
```

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/mdk/hello_world_cm4.uvmpw
```

Build both applications separately by clicking the **Rebuild** button. Build the application for the auxiliary core (cm0plus) first because the primary core application project (cm4) must know the auxiliary core application binary when running the linker. It is not possible to finish the primary core linker when the auxiliary core application binary is not ready.

Run a multicore example application   The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example application**. These steps are common for both single-core and dual-core applications in µVision.

Both the primary and the auxiliary image is loaded into the device flash memory. After clicking the "Run" button, the primary core application is executed. During the primary core code execution, the auxiliary core is released from the reset. The hello_world multicore application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.
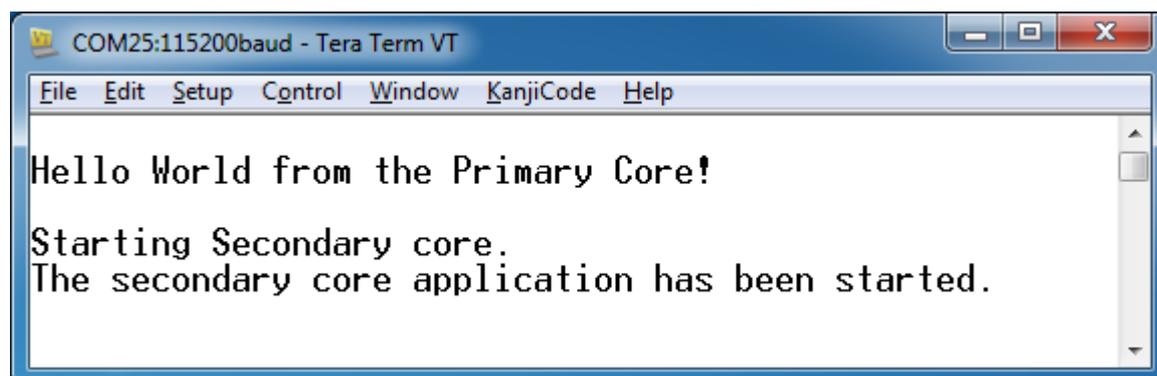
An LED controlled by the auxiliary core starts flashing indicating that the auxiliary core has been released from the reset and is running correctly.

Attach the running application of the auxiliary core by opening the auxiliary core project in the second µVision instance and clicking the "Start/Stop Debug Session" button. After this, the second debug session is opened and the auxiliary core application can be debugged.
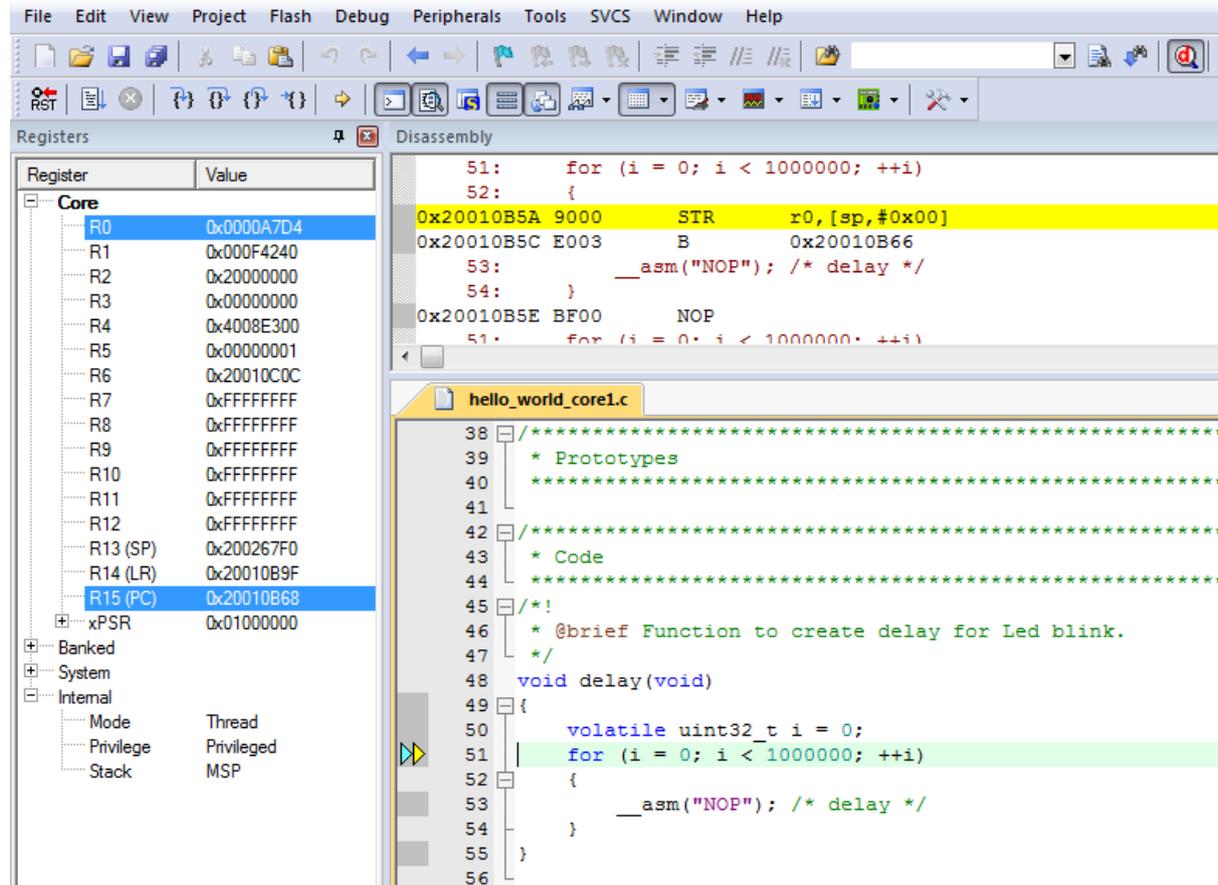
Arm describes multicore debugging using the NXP LPC54114 Cortex-M4/M0+ dual-core processor and Keil uVision IDE in Application Note 318 at www.keil.com/appnotes/docs/apnt_318.asp. The associated video can be found here.

Build a TrustZone example application   This section describes the particular steps that must be done in order to build and run a TrustZone application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/<application_name>_ns/
↪mdk
```

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/<application_name>_s/
↪mdk
```

Begin with a simple TrustZone version of the Hello World application. The TrustZone Hello World Keil MSDK/µVision workspaces are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/mdk/hello_world_
↪ns.uvmpw
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world_s.
↪uvmpw
```

> <install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.
> ↪uvmpw

This project hello_world.uvmpw contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another.

Build both applications separately by clicking **Rebuild**. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library is running the linker. It is not possible to finish the non-secure project linker with the secure project because CMSE library is not ready.

Run a TrustZone example application The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

To download and run the TrustZone application, switch to the secure application project and perform steps as described in **Run an example application**. These steps are common for single core, dual-core, and TrustZone applications in µVision. After clicking **Download and Debug**, both the secure and non-secure images are loaded into the device flash memory, and the secure application is executed. It stops at the main() function.
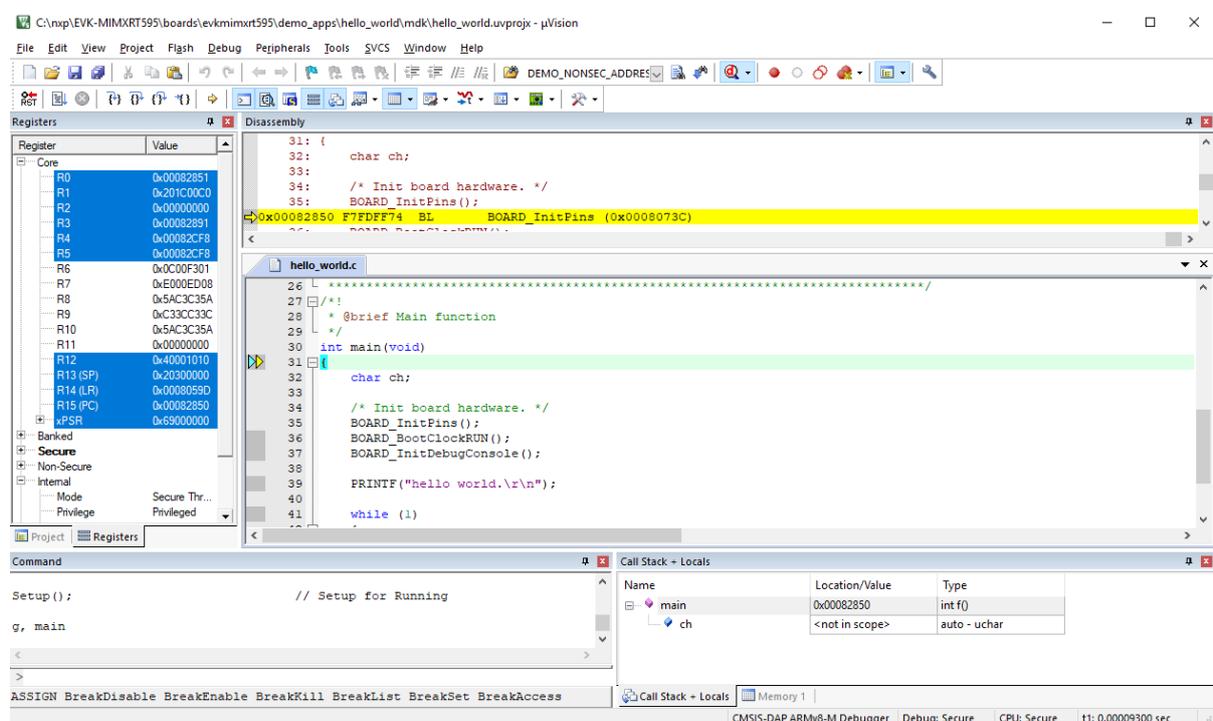
Run the code by clicking **Run** to start the application.

The hello_world application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.
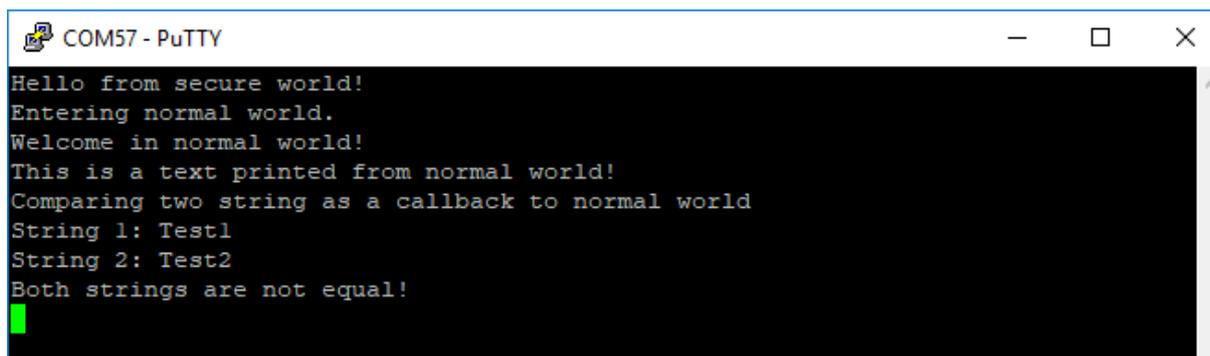
Run a demo using ARMGCC / VSCODE   This section describes the steps to run an example application from the SDK archive using the ARMGCC / VSCODE toolchain.

Refer to the *running a demo using MCUXpresso VSC* section for detailed instructions on setting up and configuring your project in Visual Studio Code.

Refer to the *CLI* section for detailed instructions on building and running your project from the command line.

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 24.12.00 or later.

Following table describes the tools included in the MCUXpresso Config Tools.

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

MCUXpresso Config Tools can be accessed in the following products:

- **Integrated** in the MCUXpresso IDE. Config tools are integrated with both compiler and debugger which makes it the easiest way to begin the development.

- **Standalone version** available for download from www.nxp.com/mcuxpresso. Recommended for customers using IAR Embedded Workbench, Keil MDK µVision, or Arm GCC.

- **Online version** available on mcuxpresso.nxp.com. Recommended doing a quick evaluation of the processor or use the tool without installation.

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

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, onboard debug interface, whether it is based on MCU-Link or the legacy OpenSDA, LPC-Link2, P&E Micro OSJTAG interface. To determine what your specific board ships with, see *Default debug interfaces*.

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

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

There are two ports, one is for core0 debug console and the other is for core1.

2. **Windows**: To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

   In the Device Manager, expand the **Ports (COM & LPT)** section to view the available ports. The COM port names are different for all the NXP boards.

   1. **CMSIS-DAP/mbed/DAPLink** interface:

   2. **P&E Micro**:

   3. **J-Link**:
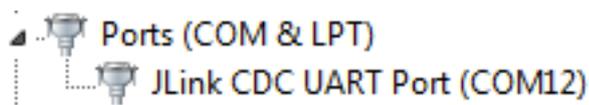
   4. **P&E Micro OSJTAG**:

   5. **MRB-KW01**:
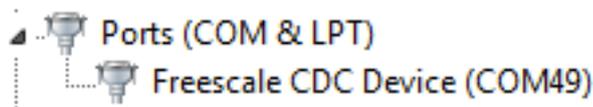
On-board Debugger   This section describes the on-board debuggers used on NXP development boards.

On-board debugger MCU-Link   MCU-Link is a powerful and cost effective debug probe that can be used seamlessly with MCUXpresso IDE, and is also compatible with 3rd party IDEs that support CMSIS-DAP protocol. MCU-Link also includes a USB to UART bridge feature (VCOM) that can be used to provide a serial connection between the target MCU and a host computer. MCU-Link features a high-speed USB interface for high performance debug. MCU-Link is compatible with Windows, MacOS and Linux. A free utility from NXP provides an easy way to install firmware updates.

On-board MCU-Link debugger supports CMSIS-DAP and J-Link firmware. See the table in *Default debug interfaces* to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.

- If using J-Link with either a standalone debug pod or MCU-Link, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.

Updating MCU-Link firmware   This firmware in this debug interface may be updated using the host computer utility called MCU-Link. 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 reprogram 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), MCU-Link 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 that most up-to-date/compatible firmware is used with MCUXpresso IDE.

NXP provides the MCU-Link utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto MCU-Link or NXP boards. The utility can be downloaded from MCU-Link.

These steps show how to update the debugger firmware on your board for Windows operating system.

1. Install the MCU-Link utility.

2. Unplug the board's USB cable.

3. Make the DFU link (install the jumper labeled 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 MCU-Link installation directory (<MCU-Link install dir>).

   1. To program CMSIS-DAP debug firmware: <MCU-Link install dir>/scripts/ program_CMSIS

   2. To program J-Link debug firmware: <MCU-Link install dir>/scripts/program_JLINK

6. Remove DFU link (remove the jumper installed in Step 3).

7. Repower the board by removing the USB cable and plugging it in again.

On-board debugger LPC-Link   LPC-Link 2 is an extensible debug probe that can be used seamlessly with MCUXpresso IDE, and is also compatible with 3rd party IDEs that support CMSIS-DAP protocol. MCU-Link also includes a USB to UART bridge feature (VCOM) that can be used to provide a serial connection between the target MCU and a host computer. LPC-Link 2 is compatible with Windows, MacOS and Linux. A free utility from NXP provides an easy way to install firmware updates.

On-board LPC-Link 2 debugger supports CMSIS-DAP and J-Link firmware. See the table in *Default debug interfaces* to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.

- If using J-Link with either a standalone debug pod or MCU-Link, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.

Updating LPC-Link 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 LPCScrypt. 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 reprogram 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 that most up-to-date/compatible firmware is used with MCUXpresso IDE.

NXP provides the LPCScrypt 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 LPCScrypt.

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in LPCScrypt user guide (LPCScrypt, select **LPCScrypt**, 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 labeled 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 LPCScrypt installation directory (<LPCScrypt install dir>).

    1. To program CMSIS-DAP debug firmware: <LPCScrypt install dir>/scripts/program_CMSIS

    2. To program J-Link debug firmware: <LPCScrypt install dir>/scripts/program_JLINK

6. Remove DFU link (remove the jumper installed in Step 3).

7. Repower the board by removing the USB cable and plugging it in again.

On-board debugger OpenSDA   OpenSDA/OpenSDAv2 is a serial and debug adapter that is built into several NXP evaluation boards. It provides a bridge between your computer (or other USB host) and the embedded target processor, which can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

The difference is the firmware implementation: OpenSDA: Programmed with the proprietary P&E Micro developed bootloader. P&E Micro is the default debug interface app. OpenSDAv2: Programmed with the open-sourced CMSIS-DAP/mbed bootloader. CMSIS-DAP is the default debug interface app.

See the table in *Default debug interfaces* to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.

- For boards with a P&E Micro interface, see PE micro to download and install the P&E Micro Hardware Interface Drivers package.

Updating OpenSDA firmware   Any NXP hardware platform that comes with an OpenSDA-compatible debug interface has the ability to update the OpenSDA firmware. This typically means to switch from the default application (either CMSIS-DAP or P&E Micro) to a SEGGER J-Link. This section contains the steps to switch the OpenSDA firmware to a J-Link interface. However, the steps can be applied to restoring the original image also. For reference, OpenSDA firmware files can be found at the links below:

- J-Link: Download appropriate image from www.segger.com/opensda.html. Choose the appropriate J-Link binary based on the table in *Default debug interfaces*. Any OpenSDA v1.0 interface should use the standard OpenSDA download (in other words, the one with no version). For OpenSDA 2.0 or 2.1, select the corresponding binary.

- CMSIS-DAP: CMSIS-DAP OpenSDA firmware is available at www.nxp.com/opensda.

- P&E Micro: Downloading P&E Micro OpenSDA firmware images requires registration with P&E Micro (www.pemicro.com).

Perform the following steps to update the OpenSDA firmware on your board for Windows and Linux OS users:

1. Unplug the board's USB cable.

2. Press the **Reset** button on the board. While still holding the button, plug the USB cable back into the board.

3. When the board re-enumerates, it shows up as a disk drive called **MAINTENANCE**.
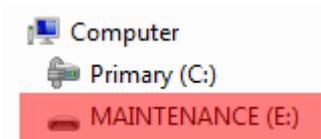
4. Drag and drop the new firmware image onto the MAINTENANCE drive.

   Note: If for any reason the firmware update fails, the board can always reenter maintenance mode by holding down **Reset** button and power cycling.

These steps show how to update the OpenSDA firmware on your board for Mac OS users.

1. Unplug the board's USB cable.

2. Press the **Reset** button of the board. While still holding the button, plug the USB cable back into the board.

3. For boards with OpenSDA v2.0 or v2.1, it shows up as a disk drive called **BOOTLOADER** in **Finder**. Boards with OpenSDA v1.0 may or may not show up depending on the bootloader version. If you see the drive in **Finder**, proceed to the next step. If you do not see the drive in Finder, use a PC with Windows OS 7 or an earlier version to either update the OpenSDA firmware, or update the OpenSDA bootloader to version 1.11 or later. The bootloader update instructions and image can be obtained from P&E Microcomputer website.

4. For OpenSDA v2.1 and OpenSDA v1.0 (with bootloader 1.11 or later) users, drag the new firmware image onto the BOOTLOADER drive in **Finder**.

5. For OpenSDA v2.0 users, type these commands in a Terminal window:

```
> sudo mount -u -w -o sync /Volumes/BOOTLOADER
> cp -X  <path to update file> /Volumes/BOOTLOADER
```

   Note: If for any reason the firmware update fails, the board can always reenter bootloader mode by holding down the **Reset** button and power cycling.

On-board debugger Multilink   An on-board Multilink debug circuit provides a JTAG interface and a power supply input through a single micro-USB connector. It is a hardware interface that allows PC software to debug and program a target processor through its debug port.

The host driver must be installed before debugging.

- See PE micro to download and install the P&E Micro Hardware Interface Drivers package.

On-board debugger OSJTAG   An on-board OSJTAG debug circuit provides a JTAG interface and a power supply input through a single micro-USB connector. It is a hardware interface that allows PC software to debug and program a target processor through its debug port.

The host driver must be installed before debugging.

- See PE micro to download and install the P&E Micro Hardware Interface Drivers package.

Default debug interfaces   The MCUXpresso SDK supports various hardware platforms that come loaded with various factory programmed debug interface configurations. The following table lists the hardware platforms supported by the MCUXpresso SDK, their default debug firmware, and any version information that helps differentiate a specific interface configuration.

Hardware platform	Default debugger firmware	On-board debugger probe
EVK-MCIMX7ULP	N/A	N/A
EVK-MIMX8MM	N/A	N/A
EVK-MIMX8MN	N/A	N/A
EVK-MIMX8MNDDR3L	N/A	N/A
EVK-MIMX8MP	N/A	N/A
EVK-MIMX8MQ	N/A	N/A
EVK-MIMX8ULP	N/A	N/A
EVK-MIMXRT1010	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1015	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1020	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1064	CMSIS-DAP	LPC-Link2
EVK-MIMXRT595	CMSIS-DAP	LPC-Link2
EVK-MIMXRT685	CMSIS-DAP	LPC-Link2
EVK9-MIMX8ULP	N/A	N/A
EVKB-IMXRT1050	CMSIS-DAP	LPC-Link2
FRDM-K22F	CMSIS-DAP	OpenSDA v2
FRDM-K32L2A4S	CMSIS-DAP	OpenSDA v2
FRDM-K32L2B	CMSIS-DAP	OpenSDA v2
FRDM-K32L3A6	CMSIS-DAP	OpenSDA v2
FRDM-KE02Z40M	P&E Micro	OpenSDA v1
FRDM-KE15Z	CMSIS-DAP	OpenSDA v2
FRDM-KE16Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z512	CMSIS-DAP	MCU-Link
FRDM-MCXA153	CMSIS-DAP	MCU-Link
FRDM-MCXA156	CMSIS-DAP	MCU-Link
FRDM-MCXA266	CMSIS-DAP	MCU-Link
FRDM-MCXA344	CMSIS-DAP	MCU-Link
FRDM-MCXA346	CMSIS-DAP	MCU-Link
FRDM-MCXA366	CMSIS-DAP	MCU-Link
FRDM-MCXC041	CMSIS-DAP	MCU-Link
FRDM-MCXC242	CMSIS-DAP	MCU-Link
FRDM-MCXC444	CMSIS-DAP	MCU-Link
FRDM-MCXE247	CMSIS-DAP	MCU-Link
FRDM-MCXE31B	CMSIS-DAP	MCU-Link
FRDM-MCXN236	CMSIS-DAP	MCU-Link
FRDM-MCXN947	CMSIS-DAP	MCU-Link
FRDM-MCXW23	CMSIS-DAP	MCU-Link

Table 1 – continued from previous page

Hardware platform	Default debugger firmware	On-board debugger probe
FRDM-MCXW71	CMSIS-DAP	MCU-Link
FRDM-MCXW72	CMSIS-DAP	MCU-Link
FRDM-RW612	CMSIS-DAP	MCU-Link
IMX943-EVK	N/A	N/A
IMX95LP4XEVK-15	N/A	N/A
IMX95LPD5EVK-19	N/A	N/A
IMX95VERDINEVK	N/A	N/A
KW45B41Z-EVK	CMSIS-DAP	MCU-Link
KW45B41Z-LOC	CMSIS-DAP	MCU-Link
KW47-EVK	CMSIS-DAP	MCU-Link
KW47-LOC	CMSIS-DAP	MCU-Link
LPC845BREAKOUT	CMSIS-DAP	LPC-Link2
LPCXpresso51U68	CMSIS-DAP	LPC-Link2
LPCXpresso54628	CMSIS-DAP	LPC-Link2
LPCXpresso54S018	CMSIS-DAP	LPC-Link2
LPCXpresso54S018M	CMSIS-DAP	LPC-Link2
LPCXpresso55S06	CMSIS-DAP	LPC-Link2
LPCXpresso55S16	CMSIS-DAP	LPC-Link2
LPCXpresso55S28	CMSIS-DAP	LPC-Link2
LPCXpresso55S36	CMSIS-DAP	MCU-Link
LPCXpresso55S69	CMSIS-DAP	LPC-Link2
LPCXpresso802	CMSIS-DAP	LPC-Link2
LPCXpresso804	CMSIS-DAP	LPC-Link2
LPCXpresso824MAX	CMSIS-DAP	LPC-Link2
LPCXpresso845MAX	CMSIS-DAP	LPC-Link2
LPCXpresso860MAX	CMSIS-DAP	LPC-Link2
MC56F80000-EVK	P&E Micro	Multilink
MC56F81000-EVK	P&E Micro	Multilink
MC56F83000-EVK	P&E Micro	OSJTAG
MCIMX93-EVK	N/A	N/A
MCIMX93-QSB	N/A	N/A
MCIMX93AUTO-EVK	N/A	N/A
MCX-N5XX-EVK	CMSIS-DAP	MCU-Link
MCX-N9XX-EVK	CMSIS-DAP	MCU-Link
MCX-W71-EVK	CMSIS-DAP	MCU-Link
MCX-W72-EVK	CMSIS-DAP	MCU-Link
MIMXRT1024-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1040-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1060-EVKB	CMSIS-DAP	LPC-Link2
MIMXRT1060-EVKC	CMSIS-DAP	MCU-Link
MIMXRT1160-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1170-EVKB	CMSIS-DAP	MCU-Link
MIMXRT1180-EVK	CMSIS-DAP	MCU-Link
MIMXRT685-AUD-EVK	CMSIS-DAP	LPC-Link2
MIMXRT700-EVK	CMSIS-DAP	MCU-Link
RD-RW612-BGA	CMSIS-DAP	MCU-Link
TWR-KM34Z50MV3	P&E Micro	OpenSDA v1
TWR-KM34Z75M	P&E Micro	OpenSDA v1
TWR-KM35Z75M	CMSIS-DAP	OpenSDA v2
TWR-MC56F8200	P&E Micro	OSJTAG
TWR-MC56F8400	P&E Micro	OSJTAG

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

Repository-Layout SDK Package

Development Tools Installation This guide explains how to install the essential tools for development with the MCUXpresso SDK.

Quick Start: Automated Installation (Recommended) The **MCUXpresso Installer** is the fastest way to get started. It automatically installs all the basic tools you need.

1. **Download the MCUXpresso Installer** from: Dependency-Installation
2. **Run the installer** and select **"MCUXpresso SDK Developer"** from the menu
3. **Click Install** and let it handle everything automatically

Manual Installation If you prefer to install tools manually or need specific versions, follow these steps:

Essential Tools

Git - Version Control **What it does**: Manages code versions and downloads SDK repositories from GitHub.

Installation:

- Visit git-scm.com
- Download for your operating system
- Run installer with default settings
- **Important**: Make sure "Add Git to PATH" is selected during installation

Setup:

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

Python - Scripting Environment **What it does**: Runs build scripts and SDK tools.

Installation:

- Install Python **3.10 or newer** from python.org
- **Important**: Check "Add Python to PATH" during installation

West - SDK Management Tool **What it does**: Manages SDK repositories and provides build commands. The west tool is developed by the Zephyr project for managing multiple repositories.

Installation:

```
pip install -U west
```

Minimum version: 1.2.0 or newer

Build System Tools

CMake - Build Configuration **What it does**: Configures how your projects are built.

Recommended version: 3.30.0 or newer

Installation:

- **Windows**: Download .msi installer from cmake.org/download
- **Linux**: Use package manager or download from cmake.org
- **macOS**: Use Homebrew (brew install cmake) or download from cmake.org

Ninja - Fast Build System **What it does**: Compiles your code quickly.

Minimum version: 1.12.1 or newer

Installation:

- **Windows**: Usually included, or download from ninja-build.org
- **Linux**: sudo apt install ninja-build or download binary
- **macOS**: brew install ninja or download binary

Ruby - IDE Project Generation (Optional) **What it does**: Generates project files for IDEs like IAR and Keil.

When needed: Only if you want to use traditional IDEs instead of VS Code.

Installation: Follow the Ruby environment setup guide

Compiler Toolchains Choose and install the compiler toolchain you want to use:

Toolchain	Best For	Download Link	Environment Variable
ARM GCC (Recommended)	Most users, free	ARM GNU Toolchain	ARMGCC_DIR
IAR EWARM	Professional development	IAR Systems	IAR_DIR
Keil MDK	ARM ecosystem	ARM Developer	MDK_DIR
ARM Compiler	Advanced optimization	ARM Developer	ARMCLANG_DIR

Setting Up Environment Variables   After toolchain installation, set an environment variable so the build system locates it:

Windows:

```
# Example for ARM GCC installed in C:\armgcc
setx ARMGCC_DIR "C:\armgcc"
```

Linux/macOS:

```
# Add to ~/.bashrc or ~/.zshrc
export ARMGCC_DIR="/usr"  # or your installation path
```

Verify Your Installation   After installation, verify everything works by opening a terminal/command prompt and running these commands:

```
# Check each tool - you should see version numbers
git --version
python --version
west --version
cmake --version
ninja --version
arm-none-eabi-gcc --version  # (if using ARM GCC)
```

Troubleshooting Installation Issues   **"Command not found" errors**:

- The tool isn't in your system PATH
- **Solution**: Add the installation directory to your PATH environment variable

Python/pip issues:

- Try using `python3` and `pip3` instead of `python` and `pip`
- On Windows, run the Command Prompt as an Administrator

Slow downloads:

- Add timeout option: `pip install -U west --default-timeout=1000`
- Use alternative mirror: `pip install -U west -i https://pypi.tuna.tsinghua.edu.cn/simple`

Building Your First Project   This guide explains how to build and run your first SDK example project using the west build system. This applies to both GitHub Repository SDK and Repository-Layout SDK Package.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development board connected via USB
- Build tools installed per *Installation Guide*

Understanding Board Support   Use the west extension to discover available examples for your board:

```
west list_project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

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

Basic Build Process

Simple Build Build the hello_world example with default settings:

```
west build -b your_board examples/demo_apps/hello_world
```

The default toolchain is armgcc, and the build system will select the first debug target as default if no config is specified.

Specifying Configuration

```
# Release build
west build -b your_board examples/demo_apps/hello_world --config release

# Debug build (default)
west build -b your_board examples/demo_apps/hello_world --config debug
```

Alternative Toolchains

```
# IAR toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar

# Other toolchains as supported by the example
```

Multicore Applications For multicore devices, specify the core ID:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug
```

For multicore projects using sysbuild:

```
west build -b evkbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_␣
↪id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always
```

Flash an Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Start a debug session:

```
west debug -r linkserver
```

Common Build Options

Clean Build Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Dry Run   See the commands that get executed without running them:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

Device Variants   For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device DEVICE_PART_NUMBER --config␣
↪release
```

Project Configuration

CMake Configuration Only   Run configuration without building:

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

Interactive Configuration   Launch the configuration GUI:

```
west build -t guiconfig
```

Troubleshooting

Build Failures   Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Getting Help   View the help information for west build:

```
west build -h
```

Check Supported Configurations   To see available configuration options and board targets for an example, refer to the below command:

```
west list_project -p examples/demo_apps/hello_world
```

Next Steps
- Explore other examples in the SDK
- Learn about *Command Line Development* for advanced options
- Try *VS Code Development* for integrated development
- Refer *Workspace Structure* to understand the SDK layout

MCUXpresso for VS Code Development   This guide covers using MCUXpresso for VS Code extension to build, debug, and develop SDK applications with an integrated development environment.

Prerequisites

- SDK workspace initialized (GitHub Repository SDK or Repository-Layout SDK Package)
- Development tools installed per *Installation Guide*
- Visual Studio Code installed
- MCUXpresso for VS Code extension installed

Extension Installation

Install MCUXpresso for VS Code   The MCUXpresso for VS Code extension provides integrated development capabilities for MCUXpresso SDK projects. Refer to the MCUXpresso for VS Code Wiki for detailed installation and setup instructions.

SDK Import and Setup

Import Methods   The SDK can be imported in several ways. The MCUXpresso for VS Code extension supports both GitHub Repository SDK and Repository-Layout SDK Package distributions.

Import GitHub Repository SDK   Click **Import Repository** from the **QUICKSTART PANEL**
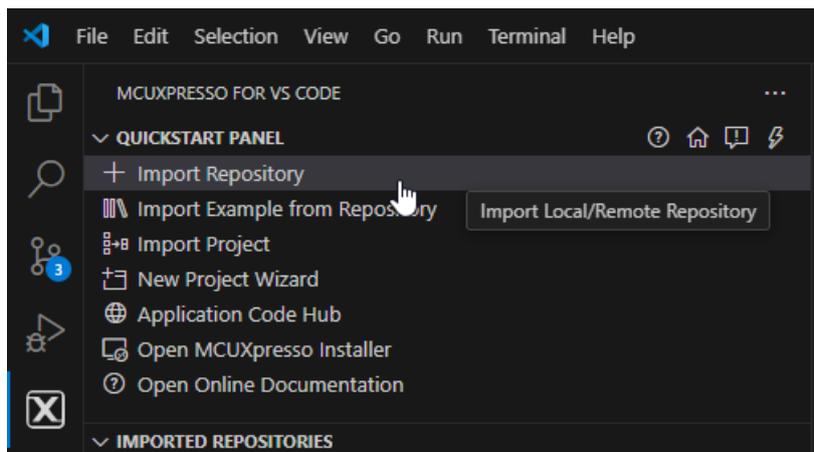
Note: You can import the SDK in several ways. Refer to MCUXpresso for VS Code Wiki for details.

Select **Local** if you've already obtained the SDK according to *setting up the repo*. Select your location and click **Import**.
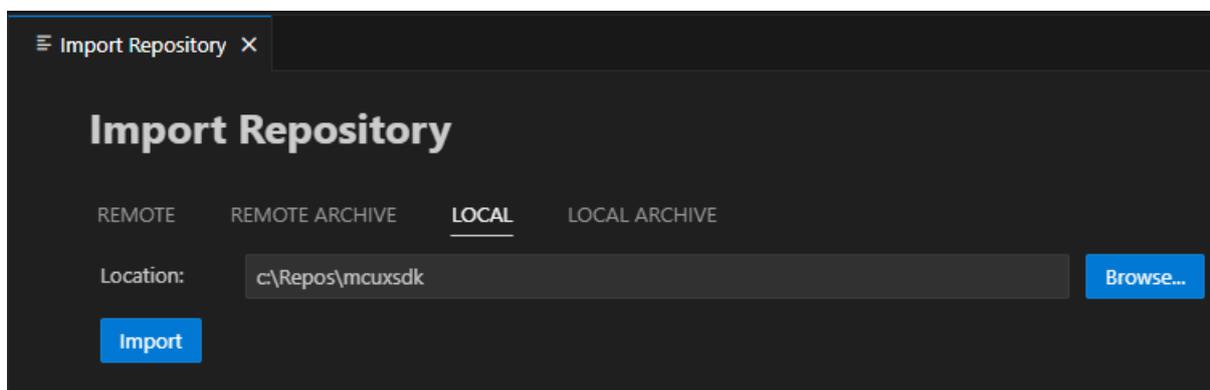
Import Repository-Layout SDK Package   Click **Import Repository** from the **QUICKSTART**
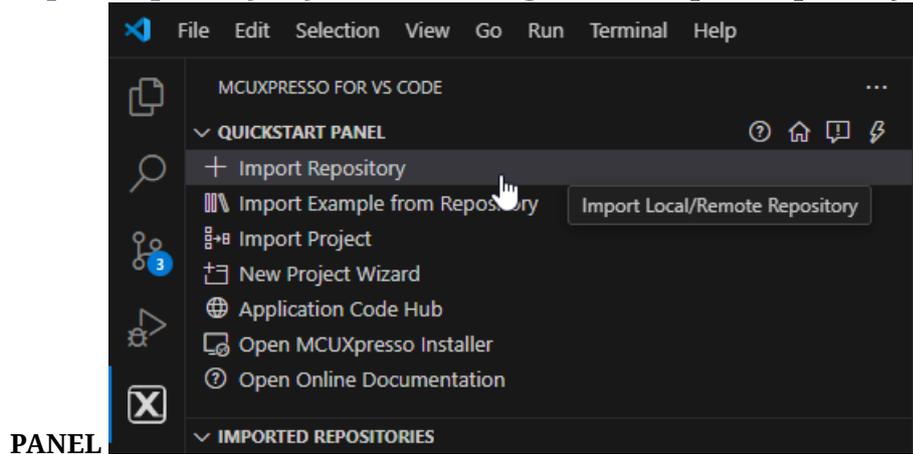
PANEL

Select **Local** if you've already unzipped the Repository-Layout SDK Package. Select your location and click **Import**.
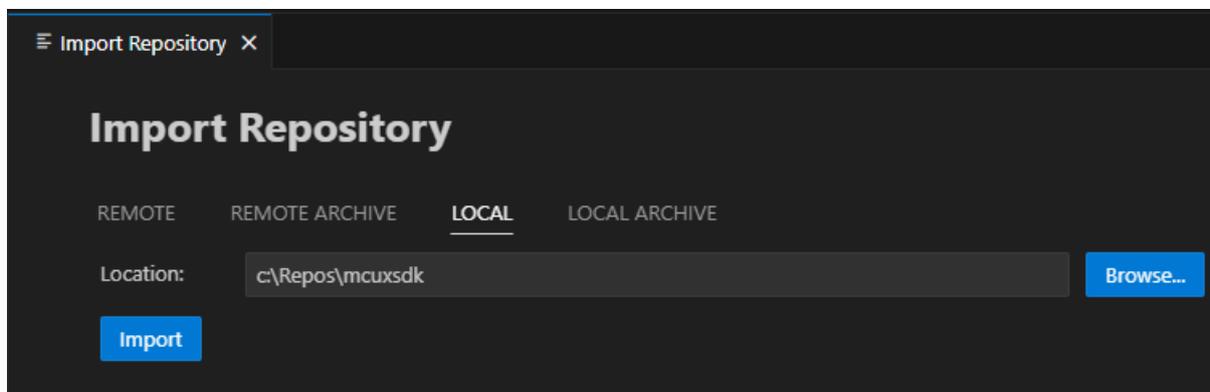
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.
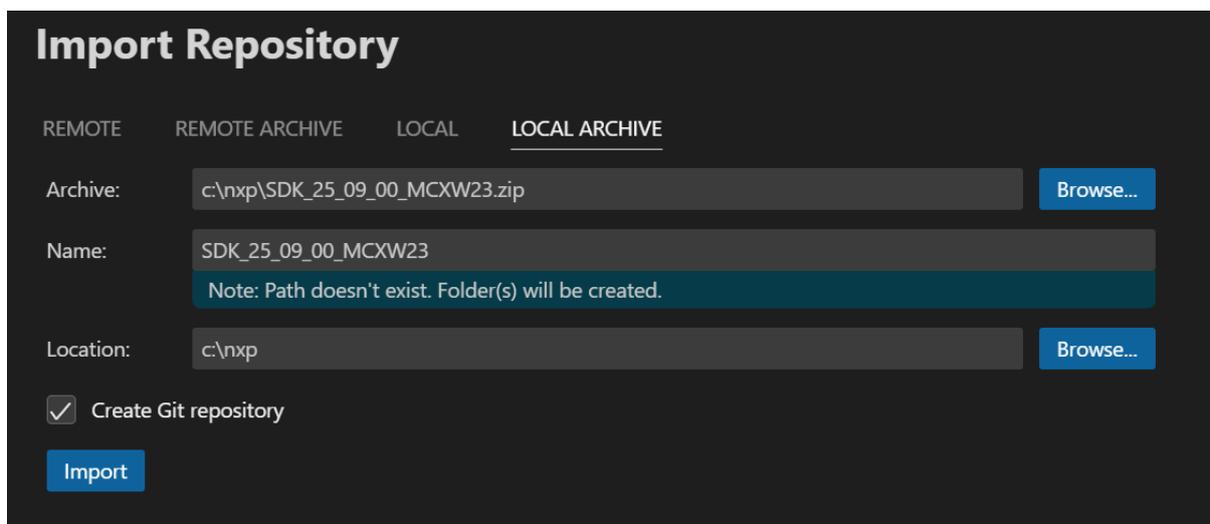
Building Example Applications

Import Example Project

1. Click **Import Example from Repository** from the **QUICKSTART PANEL**
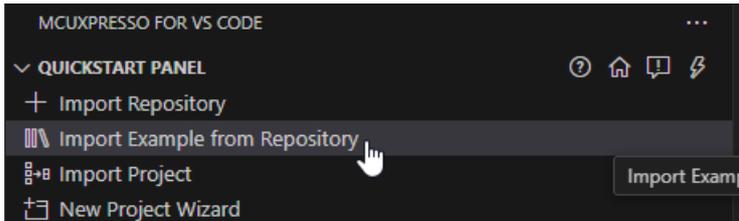
2. Configure project settings:

- **MCUXpresso SDK**: Select your imported SDK
- **Arm GNU Toolchain**: Choose toolchain
- **Board**: Select your target development board
- **Template**: Choose example category
- **Application**: Select specific example (e.g., hello_world)
- **App type**: Choose between Repository applications or Freestanding applications

3. Click **Import**
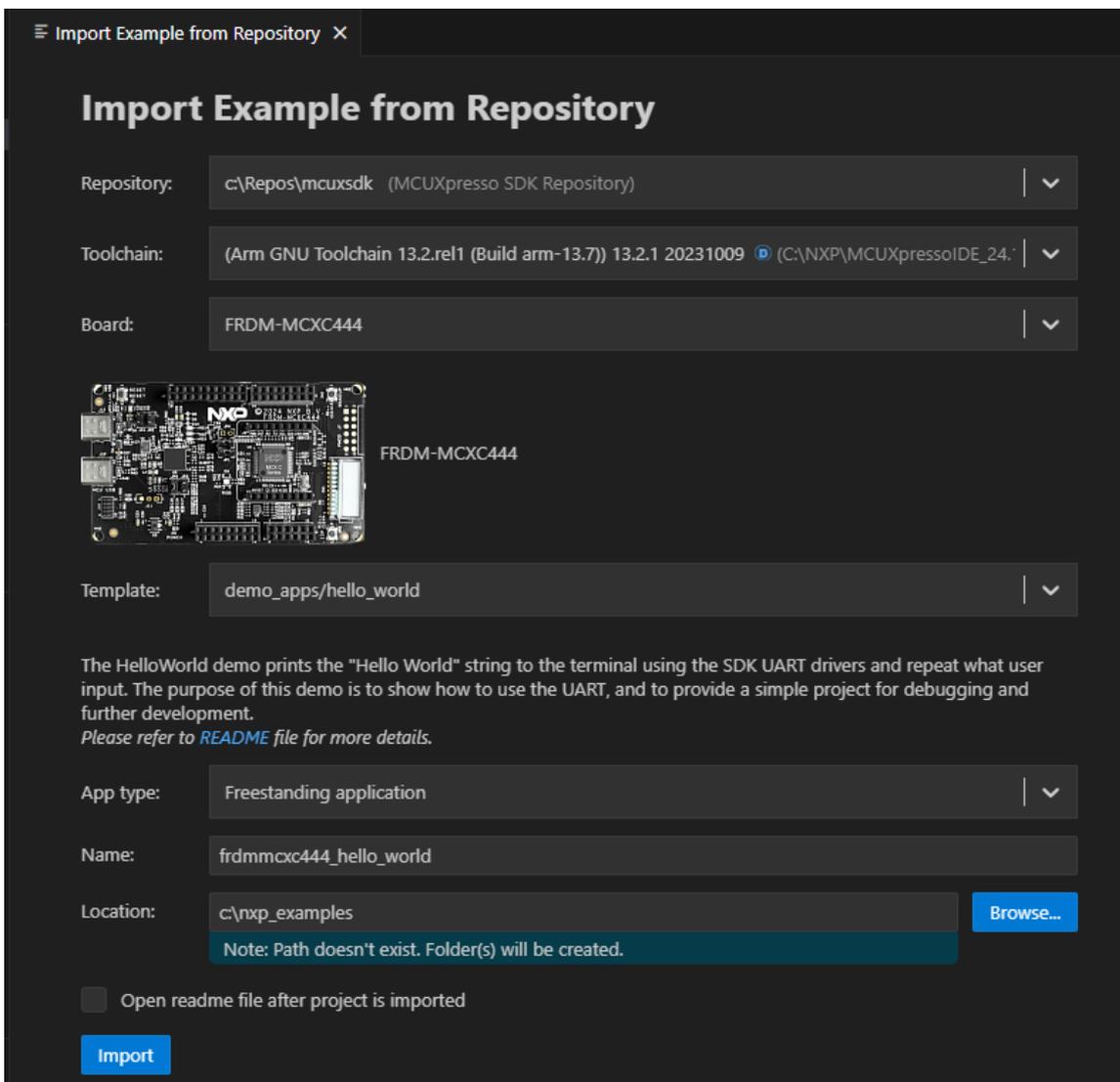
Application Types   **Repository Applications:**

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

Freestanding Applications:

- Imported to user-defined location
- Independent of SDK location

Trust Confirmation   VS Code will prompt you to confirm if the imported files are trusted. Click **Yes** to proceed.

Building Projects

Build Process

1. Navigate to **PROJECTS** view
2. Find your project
3. Click the **Build Project** icon
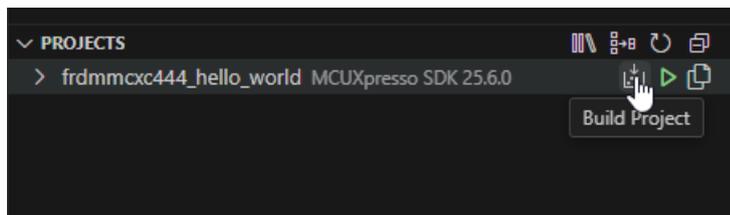
The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

1. Open **Serial Monitor** from VS Code's integrated terminal
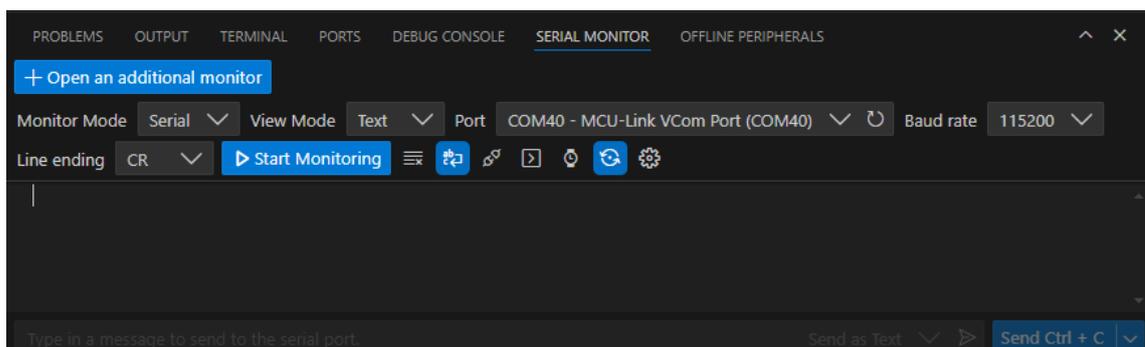
2. Configure serial settings:
   - **VCom Port**: Select port for your device
   - **Baud Rate**: Set to 115200

Debug Session

1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session
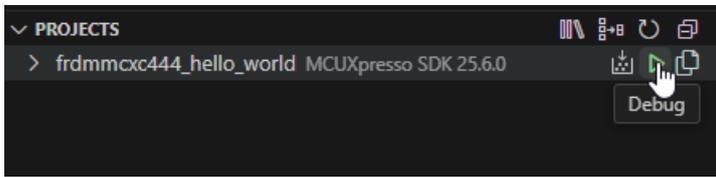
The debug session will begin with debug controls initially at the top of the interface.

Debug Controls   Use the debug controls to manage execution:

- **Continue**: Resume code execution
- **Step controls**: Navigate through code

- **Stop**: Terminate debug session

Monitor Output   Observe application output in the **Serial Monitor** to verify correct operation.
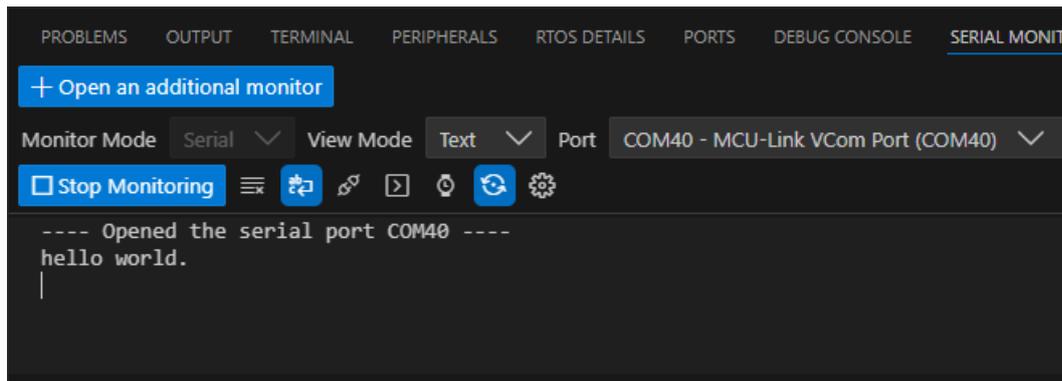
Debug Probe Support For comprehensive information on debug probe support and configuration, refer to the MCUXpresso for VS Code Wiki DebugK section.

Project Configuration

Workspace Management The extension integrates with the MCUXpresso SDK workspace structure, providing access to:

- Example applications
- Board configurations
- Middleware components
- Build system integration

Multi-Project Support The PROJECTS view allows management of multiple imported projects within the same workspace.

Troubleshooting

Import Issues **SDK not detected:**

- Verify SDK workspace is properly initialized
- Ensure all required repositories are updated
- Check SDK manifest files are present

Project import failures:

- Confirm board support exists for selected example
- Verify toolchain installation
- Check example compatibility with selected board

Build Problems **Build failures:**

- Check integrated terminal for error messages
- Verify all dependencies are installed
- Ensure toolchain is properly configured

Debug Issues   **Debug session fails:**

- Verify board connection via USB
- Check debug probe drivers are installed
- Confirm build completed successfully

Serial monitor problems:

- Verify correct VCom port selection
- Check baud rate configuration (115200)
- Ensure board drivers are installed

Integration with Command Line   MCUXpresso for VS Code integrates with the underlying west build system, allowing seamless integration with command line workflows described in *Command Line Development*.

Advanced Features

Project Types   The extension supports both repository-based and freestanding project types, providing flexibility in project organization and SDK integration.

Build System Integration   The extension leverages the MCUXpresso SDK build system, providing access to all build configurations and options available through command line tools.

Next Steps

- Explore additional examples in the SDK
- Review *Command Line Development* for advanced build options
- Refer MCUXpresso for VS Code Wiki for detailed documentation
- Learn about *SDK Architecture* for better understanding of the development environment

Command Line Development   This guide covers developing with the MCUXpresso SDK using command line tools and the west build system. This workflow applies to both GitHub Repository SDK and Repository-Layout SDK Package distributions.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development tools installed per *Installation Guide*
- Target board connected via USB

Understanding Board Support   Use the west extension to discover available examples for your board:

```
west list_project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

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

Basic Build Commands

Standard Build Process　Build with default settings (armgcc toolchain, first debug config):

```
west build -b your_board examples/demo_apps/hello_world
```

Specifying Build Configuration

```
# Release build
west build -b your_board examples/demo_apps/hello_world --config release

# Debug build with specific toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar --config debug
```

Multicore Applications　For multicore devices, specify the core ID:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug
```

For multicore projects using sysbuild:

```
west build -b evkbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always
```

Shield Support　For boards with shields:

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

Advanced Build Options

Clean Builds　Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Dry Run　See what commands would be executed:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

Device Variants　For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device MK22F12810 --config release
```

Project Configuration

CMake Configuration Only   Run configuration without building:

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

Interactive Configuration   Launch the configuration GUI:

```
west build -t guiconfig
```

Flashing and Debugging

Flash Application   Flash the built application to your board:

```
west flash -r linkserver
```

Debug Session   Start a debugging session:

```
west debug -r linkserver
```

IDE Project Generation   Generate IDE project files for traditional IDEs:

```
# Generate IAR project
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug -p always -t guiproject
```

IDE project files are generated in mcuxsdk/build/<toolchain> folder.

Note: Ruby installation is required for IDE project generation. See *Installation Guide* for setup instructions.

Troubleshooting

Build Failures   Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Toolchain Issues   Verify environment variables are set correctly:

```
# Check ARM GCC
echo $ARMGCC_DIR
arm-none-eabi-gcc --version

# Check IAR (if using)
echo $IAR_DIR
```

Getting Help   Display help information:

```
west build -h
west flash -h
west debug -h
```

Check Supported Configurations   If unsure about supported options for an example:

```
west list_project -p examples/demo_apps/hello_world
```

Best Practices

Project Organization

- Keep custom projects outside the SDK tree
- Use version control for your application code
- Document any SDK modifications

Build Efficiency

- Use `-p always` for clean builds when troubleshooting
- Leverage `--dry-run` to understand build processes
- Use specific configs and toolchains to reduce build time

Development Workflow

1. Start with existing examples closest to your requirements
2. Copy and modify rather than building from scratch
3. Test with hello_world before moving to complex examples
4. Use configuration tools for pin muxing and clock setup

Next Steps

- Explore *VS Code Development* for integrated development experience
- Review *Workspace Structure* to understand SDK organization
- Refer build system documentation for advanced configurations

Workspace Structure   After you initialize your SDK workspace, it creates a specific directory structure that organizes all SDK components. This structure is identical for both GitHub Repository SDK and Repository-Layout SDK Package.

Top-Level Organization

```
your-sdk-workspace/
    manifests/          # West manifest repository
    mcuxsdk/            # Main SDK content
```

The `mcuxsdk/` directory serves as your primary working directory and contains all the SDK components.

SDK Component Layout Based on the actual SDK structure, the main directories include:

Directory	Contents	Purpose
arch/	Architecture-specific files	ARM CMSIS, build configurations
cmake	Build system modules	CMake configuration and build rules
compo	Software components	Reusable software libraries and utilities
devices	Device support packages	MCU-specific headers, startup code, linker scripts
drivers	Peripheral drivers	Hardware abstraction layer for MCU peripherals
examp	Sample applications	Demonstration code and reference implementations
middle	Optional software stacks	Networking, graphics, security, and other libraries
rtos/	Operating system support	FreeRTOS integration
scripts	Build and utility scripts	West extensions and development tools
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.	

Example Organization Examples follow a two-tier structure separating common code from board-specific implementations:

Common Example Files

```
examples/demo_apps/hello_world/
   CMakeLists.txt        # Build configuration
   example.yml           # Example metadata
   hello_world.c         # Application source code
   Kconfig               # Configuration options
   readme.md             # General documentation
```

Board-Specific Files

```
examples/_boards/your_board/demo_apps/hello_world/
   app.h                    # Board specific application header
   example_board_readme.md  # Board specific documentation
   hardware_init.c          # Board specific hardware initialization
   pin_mux.c                # Pin multiplexing configuration
   pin_mux.h                # Pin multiplexing header definitions
   hello_world.bin          # Pre-built binary for quick testing
   hello_world.mex          # MCUXpresso Config Tools project file
   prj.conf                 # Board specific Kconfig configuration
   reconfig.cmake           # Board specific cmake configuration overrides
```

Device Support Structure   Device support is organized hierarchically by MCU family:

```
devices/
  MCX/                # MCU portfolio
    MCXW/             # MCU family
      MCXW235/        # Specific device
        MCXW235.h         # Device register definitions
        drivers/        # Device-specific drivers
        gcc/            # GNU toolchain files
        iar/            # IAR toolchain files
        mcuxpresso/       # MCUXpresso IDE files
        startup_MCXW235.c # Startup and vector table
        system_MCXW235.c  # System initialization
```

Middleware Organization   Middleware components are categorized by functionality and maintained in separate repositories. Based on the manifest files, common middleware categories include:

- **Connectivity**: USB, TCP/IP, industrial protocols
- **Security**: Cryptographic libraries, secure boot
- **Wireless**: Bluetooth, IEEE 802.15.4, Wi-Fi
- **Graphics**: Display drivers, UI frameworks
- **Audio**: Processing libraries, voice recognition
- **Machine Learning**: Inference engines, neural networks
- **Safety**: IEC60730B safety libraries
- **Motor Control**: Motor control and real-time control libraries

Documentation Structure   SDK documentation is distributed across multiple locations:

- docs/ - Core SDK documentation and build infrastructure
- Component repositories - API documentation and integration guides
- Board directories - Hardware-specific setup instructions

For complete documentation, refer to the online documentation.

Understanding Example Structure   Each example has **two README files**:

1. General README: examples/demo_apps/hello_world/readme.md

- What the example does
- General functionality description
- Common usage information

2. Board-Specific README: examples/_boards/{board_name}/demo_apps/hello_world/example_board_readme.md

- Board-specific setup instructions
- Hardware connections required
- Board-specific behavior notes

Tip: Always check both readme files - start with the general one, then read the board-specific one for detailed setup.

1.3   Getting Started with MCUXpresso SDK GitHub

1.3.1   Getting Started with MCUXpresso SDK Repository

Welcome to the **GitHub Repository SDK Guide**. This documentation provides instructions for setting up and working with the MCUXpresso SDK distributed in a **multi-repository model**. The SDK is distributed across multiple GitHub repositories and managed using the **Zephyr West** tool, enabling modular development and streamlined workflows.

Overview

The GitHub Repository SDK approach offers:

- **Modular Structure**: Multiple repositories for flexibility and scalability.
- **Zephyr West Integration**: Simplified repository management and synchronization.
- **Cross-Platform Support**: Designed for MCUXpresso SDK development environments.

Benefits of the Multi-Repository Approach

- **Scalability**: Easily add or update components without impacting the entire SDK.
- **Collaboration**: Enables distributed development across teams and repositories.
- **Version Control**: Independent versioning for components ensures better stability.
- **Automation**: Zephyr West simplifies dependency handling and repository synchronization.

Setup and Configuration

Follow these steps to prepare your development environment:

GitHub Repository Setup   This guide explains how to initialize your MCUXpresso SDK workspace from GitHub repositories using the west tool. The GitHub Repository SDK uses multiple repositories hosted on GitHub to provide modular, flexible development.

Prerequisites   Verify the requirements:

System Requirements:

- Python 3.8 or later
- Git 2.25 or later
- CMake 3.20 or later
- Build tools for your target platform

Verification Commands:

```
python --version    # Should show 3.8+
git --version       # Should show 2.25+
cmake --version     # Should show 3.20+
west --version      # Should show west tool installation
```

Workspace Initialization   The GitHub Repository SDK uses the Zephyr west tool to manage multiple repositories containing different SDK components.

Step 1: Initialize Workspace   Create and initialize your SDK workspace from GitHub:

Get the latest SDK from main branch:

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk
```

Get SDK at specific revision:

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk --mr {revision}
```

Note: Replace {revision} *with the desired release tag, such as* v25.09.00

Step 2: Choose Your Repository Update Strategy   Navigate to the SDK workspace:

```
cd mcuxpresso-sdk
```

The west tool manages multiple GitHub repositories containing different SDK components. You have two options for downloading:

Option A: Download All Repositories (Complete SDK)   Download all SDK repositories for comprehensive development:

```
west update
```

This command downloads all the repositories defined in the manifest from GitHub. Initial download takes several minutes and requires ~7 GB of disk space.

Best for:

- Exploring the complete SDK
- Multi-board development projects
- Comprehensive middleware evaluation

Option B: Targeted Repository Download (Recommended)   Download only repositories needed for your specific board or device to save time and disk space:

```
# For specific board development
west update_board --set board your_board_name

# For specific device family development
west update_board --set device your_device_name

# List available repositories before downloading
west update_board --set board your_board_name --list-repo
```

Best for:

- Single board development

- Faster setup and reduced disk usage
- Focused development workflows

Examples:

```
# Update only repositories for FRDM-MCXW23 board
west update_board --set board frdmmcxw23

# Update only repositories for MCXW23 device family
west update_board --set device mcxw23
```

Step 3: Verify Installation   Confirm successful setup:

```
# Verify workspace structure
ls -la
# Should show: manifests/ and mcuxsdk/ directories

# Test build system
west list_project -p examples/demo_apps/hello_world
# Should display available build configurations
```

Advanced Repository Management   The west extension command `update_board` provides advanced repository management capabilities for optimized workspace setup with GitHub repositories.

Board-Specific Setup   Update only repositories required for a specific board:

```
# Update only repositories for specific board, e.g., frdmmcxw23
west update_board --set board frdmmcxw23

# List available repositories for the board before updating
west update_board --set board frdmmcxw23 --list-repo
```

Device-Specific Setup   Update only repositories required for a specific device family:

```
# Update only repositories for specific device, e.g., MCXW235
west update_board --set device mcxw23

# List available repositories for the device family
west update_board --set device mcxw23 --list-repo
```

Custom Configuration   For advanced users who want to create custom repository combinations:

```
# Use custom configuration file
west update_board --set custom path/to/custom-config.yml

# Generate custom configuration template
cp manifests/boards/custom.yml.template my-custom-config.yml
```

Benefits of Targeted Setup   **Reduced Download Size**

- Download only components needed for your target board or device
- Significantly faster initial setup for focused development

- Typical reduction from 7 GB to 2GB

Optimized Workspace

- Cleaner workspace with relevant components only
- Reduced disk space usage
- Faster repository operations

Flexible Development

- Switch between different board configurations easily
- Maintain separate workspaces for different projects
- Include optional components as needed

Repository Information Before setting up your workspace, you can explore what repositories are available:

```
# Display repository information in console
west update_board --set board frdmmcxw23 --list-repo

# Export repository information to YAML file for reference
west update_board --set board frdmmcxw23 --list-repo -o board-repos.yml
```

This command lists all the available repositories with descriptions and outlines the included components in the workspace.

Package Generation (Optional) The `update_board` command can also generate ZIP packages for offline distribution:

```
# Generate board-specific SDK package
west update_board --set board frdmmcxw23 -o frdmmcxw23-sdk.zip
```

Note: Package generation is primarily intended for creating custom SDK distributions. For regular development, use the workspace update commands without the -o option.

Workspace Management

Updating Your Workspace Keep your SDK current with latest updates from GitHub:

For Complete SDK Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update all component repositories
cd ..
west update
```

For Targeted Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update board-specific repositories
cd ..
west update_board --set board your_board_name
```

Workspace Status   Check workspace synchronization status:

```
# Show status of all repositories
west status

# Show detailed information about repositories
west list
```

Troubleshooting   **Network Issues:**

- Use `west update --keep-descendants` for partial failures
- Configure Git credentials for private repositories
- Check firewall settings for Git protocol access

Permission Issues:

- Ensure write permissions in workspace directory
- Run commands without sudo/administrator privileges
- Verify Git SSH key configuration for authenticated access

Disk Space:

- Full SDK workspace requires approximately 7-8 GB
- Targeted workspace typically requires 1-2 GB
- Use board-specific setup to reduce workspace size

Repository Management Issues:

- Verify board/device names match available configurations
- Check that custom YAML files follow the correct template format
- Use `--list-repo` to verify available repositories before setup

Next Steps   With your workspace initialized:

1. Review *Workspace Structure* to understand the layout
2. Build your first project with *First Build Guide*
3. Explore *Development Workflows MCUXPresso VSCode* or *Development Workflows Command Line* for the details on project setup and execution

For advanced repository management, see the west tool documentation.

Explore SDK Structure and Content

Learn about the organization of the SDK and its components:

SDK Architecture Overview   The MCUXpresso SDK uses a modular architecture where software components are distributed across multiple repositories hosted on GitHub and managed through the west tool. This approach provides flexibility, maintainability, and enables selective component inclusion.

Repository Organization   Based on the manifest structure, the SDK consists of four main repository categories:

Manifest Repository   The manifest repo (mcuxsdk-manifests) contains the west.yml manifest file that tracks all other repositories in the SDK.

Base Repositories   Recorded in submanifests/base.yml and loaded in the root west.yml manifest file. These are the foundational repositories that build the SDK:

- **Devices**: MCU-specific support packages
- **Examples**: Demonstration applications and code samples
- **Boards**: Board support packages

Middleware Repositories   Recorded in the submanifests/middleware subdirectory, categorized according to functionality:

- **Connectivity**: Networking stacks, USB, and communication protocols
- **Security**: Cryptographic libraries and secure boot components
- **Wireless**: Bluetooth, IEEE 802.15.4, and other wireless protocols
- **Graphics**: Display drivers and UI frameworks
- **Audio**: Audio processing and voice recognition libraries
- **Machine Learning**: AI inference engines and neural network libraries
- **Safety**: IEC60730B safety libraries
- **Motor Control**: Motor control and real-time control libraries

Internal Repositories   Recorded in submanifests/internal.yml and grouped into the "bifrost" group. These are only visible to NXP internal developers and hosted on NXP internal git servers.

Repository Hosting   Public repositories are hosted on GitHub under these organizations:

- nxp-mcuxpresso
- NXP
- nxp-zephyr

Internal repositories are hosted on NXP's private Git infrastructure.

Benefits of This Architecture   **Selective Integration**: Projects include only required components, reducing memory footprint and build complexity.

Independent Versioning: Each component maintains its own release cycle and version control.

Community Collaboration: Public repositories accept community contributions through standard Git workflows.

Scalable Maintenance: Component owners can update their repositories without affecting the entire SDK.

Workspace Management   The west tool manages repository synchronization, version tracking, and workspace updates. All repositories are checked out under the mcuxsdk/ directory with their designated paths defined in the manifest files.

Development Workflows

Get started with building and running projects:

Using MCUXpresso Config Tools MCUXpresso Config tools provide a user-friendly way to configure hardware initialization of your projects. This guide explains the basic workflow with the MCUXpresso SDK west build system and the Config Tools.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- MCUXpresso Config Tools standalone installed (version 25.09 or above)
- MCUXpresso SDK Project that can be successfully built

Board Files MCUXpresso Config Tools generate source files for the board. These files include pin_mux.c/h and clock_config.c/h. The files contain initialization code functions that reflect the hardware configuration in the Config Tools. Within the SDK codebase, these files are specific for the board and either shared by multiple example projects or specific for one example. Open or import the configuration from the SDK project in the Config Tools and customize the settings to match the custom board or specific project use case and regenerate the code. See *User Guide for MCUXpresso Config Tools (Desktop)* (document GSMCUXCTUG) for details.

Note: When opening the configuration for SDK example projects, the board files may be shared across multiple examples. To ensure a separate copy of the board configuration files exists, create a freestanding project with copied board files.

Visual Studio Code To open the configuration in Visual Studio Code, use the context menu for the project to access Config Tools. See MCUXpresso Extension Documentation for details. Otherwise, use the manual workflow described in detail in the following section.

Manual Workflow Use the following steps:

1. Before using Config Tools, run the west command to get the project information for Config Tools from the SDK project files, for example:

   ```
   west cfg_project_info -b lpcxpresso55s69 …mcuxsdk/examples/demo_apps/hello_world/ -Dcore_
   ↪id=cm33_core0
   ```

   This results in the creation of the project information json file that is searched by the config tools when the configuration is created. The parameters of the command should match the build parameters that will be used for the project.

2. Launch the MCUXpresso Config Tools and in the **Start development** wizard, select **Create a new configuration based on the existing IDE/Toolchain project**. Select the created "cfg_tools" subfolder as a project folder (for example: …mcuxsdk/examples/demo_apps/hello_world/cfg_tools/).

Updating the SDK West project **Note:** Updating project is supported with Config Tools V25.12 or newer only.

Changes in the Config tools generated source code modules may require adjustments to the toolchain project to ensure a successful build. These changes may mean, for example, adding the newly generated files, adding include paths, required drivers, or other SDK components.

This section describes how to manually resolve the changes needed in the project within the toolchain projects based on the SDK project managed by the West tool.

After the configuration in the Config Tools is finished, write updated files to the disk using the 'Update Code' command. The written files include a json file with the required changes for the toolchain project.

To resolve the changes in the project in the terminal, launch the west command that updates the project. For example:

```
west cfg_resolve -b lpcxpresso55s69 ...mcuxsdk/examples/demo_apps/hello_world/ -Dcore_id=cm33_core0
```

This command updates the appropriate cmake and kconfig files to address the changes. After this, the application can be built.

Note: The cfg_resolve command supports additional arguments. Launch the *west cfg_resolve -h* command to get the list and description.

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

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

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

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

MCUXpresso SDK

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

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

Development tools

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

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

Supported development systems

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

Development boards	MCU devices			
LPCXpresso	LPC54005JBD100, LPC54016JET100, LPC54S005JBD100, LPC54S016JET100,	LPC54005JET100, LPC54016JET180, LPC54S005JET100, LPC54S016JET180,	LPC54016JBD100, LPC54018JBD208, LPC54S016JBD100, LPC54S018JBD208,	LPC54016JBD208, LPC54018JET180, LPC54S016JBD208, **LPC54S018JET180**

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.

MCU Boot MCU Boot (formerly KBOOT) NXP/Freescale proprietary loader

coreHTTP coreHTTP

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

SDMMC stack The SDMMC software is integrated with MCUXpresso SDK to support SD/MMC/SDIO standard specification. This also includes a host adapter layer for bare-metal/RTOS applications.

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

mbedTLS mbedtls SSL/TLS library v2.x

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

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

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

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

LittleFS LittleFS filesystem stack

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

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

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

AWS IoT Amazon Web Service (AWS) IoT Core SDK.

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.

Debug SRAM target with J-Link on MCUXpresso IDE issue

If using the J-Link to Debug SRAM target on the IDE, when clicking the Restart button, the debug fails. The issue is caused because the integrity of the SRAM image cannot be guaranteed after it has been executed, and the MCUXpresso IDE does not reload the code before restarting. Plain

load images (images loaded from external SPI flash to SRAMX by the ROM bootloader then executed) cannot be debugged using MCUXpresso version 11.0.0. Code should be developed using the SRAM debug method, then programmed into flash once debugging is completed. For more information, see the Getting Started with MCUXpresso SDK User's Guide (document MCUXSDKG-SUG).

Example freertos_spi fails in MCUXpresso IDE

The example freertos_spi fails in the MCUXpresso IDE environment.

USB high-speed interrupt endpoint issue

If the user wants to use a high-speed interrupt endpoint, the maximum packet size should be 512 bytes.

Example freertos_spi fails in MCUXpresso IDE

The example freertos_spi fails in the MCUXpresso IDE environment.

The usart_wakeup_deepsleep doesn't work

The example doesn't work.

Affected toolchains: mcux **Affected platforms:** lpcxpresso54s018

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.

AES

[2.0.3]

- Edit aes_one_block() function to be interrupt safe.

[2.0.2]

- Fix MISRA-2012 issues.

[2.0.1]

- Improvements

  - GCM constant time tag comparison.

[2.0.0]

- Initial version.

CLOCK

[2.3.3]

- Improvements

  - Added lost comments for some enumerations.

[2.3.2]

- Bug Fixes

  - Fixed violation of MISRA C-2012 rule 5.7

[2.3.1]

- Bug Fixes

  - Fixed MISRA C-2012 rule 10.1, rule 10.4, rule 10.8, rule 15.5 and so on.
  - Fixed IAR warning Pa082 for the clock driver.

[2.3.0]

- New feature:

    - Moved SDK_DelayAtLeastUs function from clock driver to common driver.

[2.2.0]

- New Feature:

    - add new APIs including CLOCK_GetEmcClkFreq and CLOCK_GetMCanClkFreq due to removing some variables in enum clock_name_t

[2.1.0]

- Bug Fix:

    - Fix flexcomm0-9 clock calculation.

    - Correct the return frequency of CLOCK_GetFrgClkFreq.

    - Fix the bug in function CLOCK_GetPllConfig() to refine the cache feature.

    - Update the code to suppress the incorrect configuration in CLOCK_GetUsbPLLOutFromSetup().

    - Fix C++ build errors in CLOCK_GetClockAttachId() and CLOCK_AttachClk().

- New feature

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

[2.0.4]

- Bug Fix:

    - Fix attach incorrect attach_id.

[2.0.3]

- New Feature:

    - add get actual clock attach id api to allow users to obtain the actual clock source in target register.

- Bug Fix:

    - The attach clock and get actual clock attach id apis should check combination of two clock source.

- Optimization:

    - Make the judgement statments more clear.

    - Strengthen the compatibility of clock attatch id.

    - Remove some unmeaningful definitions and add some useful ones to enhance readability.

[2.0.2]

- Change CLOCK_SetupFROClocking from a macro to a function for different FRO setting address per different ROM version.

[2.0.1]

- some minor fixes.

[2.0.0]

- initial version.

COMMON

[2.6.3]

- Bug Fixes
  - Fixed build issue of CMSIS PACK BSP example caused by CMSIS 6.1 issue.

[2.6.2]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule for implicit conversions in boolean contexts

[2.6.1]

- Improvements
  - Support Cortex M23.

[2.6.0]

- Bug Fixes
  - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

- Improvements

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

[2.4.0]

- New Features

    - Added EnableIRQWithPriority, IRQ_SetPriority, and IRQ_ClearPendingIRQ for ARM.

    - Added MSDK_EnableCpuCycleCounter, MSDK_GetCpuCycleCount for ARM.

[2.3.3]

- New Features

    - Added NETC into status group.

[2.3.2]

- Improvements

    - Make driver aarch64 compatible

[2.3.1]

- Bug Fixes

    - Fixed MAKE_VERSION overflow on 16-bit platforms.

[2.3.0]

- Improvements

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

[2.2.10]

- Bug Fixes

    - Fixed the ATOMIC macros build error in cpp files.

[2.2.9]

- Bug Fixes

    - Fixed MISRA C-2012 issue, 5.6, 5.8, 8.4, 8.5, 8.6, 10.1, 10.4, 17.7, 21.3.

    - Fixed SDK_Malloc issue that not allocate memory with required size.

[2.2.8]

- Improvements

    - Included stddef.h header file for MDK tool chain.

- New Features:

    - Added atomic modification macros.

[2.2.7]

- Other Change

    – Added MECC status group definition.

[2.2.6]

- Other Change

    – Added more status group definition.

- Bug Fixes

    – Undef __VECTOR_TABLE to avoid duplicate definition in cmsis_clang.h

[2.2.5]

- Bug Fixes

    – Fixed MISRA C-2012 rule-15.5.

[2.2.4]

- Bug Fixes

    – Fixed MISRA C-2012 rule-10.4.

[2.2.3]

- New Features

    – Provided better accuracy of SDK_DelayAtLeastUs with DWT, use macro SDK_DELAY_USE_DWT to enable this feature.

    – Modified the Cortex-M7 delay count divisor based on latest tests on RT series boards, this setting lets result be closer to actual delay time.

[2.2.2]

- New Features

    – Added include RTE_Components.h for CMSIS pack RTE.

[2.2.1]

- Bug Fixes

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

[2.2.0]

- New Features

    – Moved SDK_DelayAtLeastUs function from clock driver to common driver.

[2.1.4]

- New Features

    – Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
    - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.

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

[2.3.4]

- Bug Fixes
  - Fixed ERRATA ERR053024 CTIMER will enter interrupt twice when function clock much slower than bus clock.

[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

[2.5.4]

• Bug Fixes

  – Fixed coverity issues with CERT INT30-C, CERT INT31-C compliance.

[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_ 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.

DMIC

[2.3.3]

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

[2.3.2]

- New Features
    - Supported 4 channels in driver.

[2.3.1]

- Bug Fixes

    – Fixed the issue that DMIC_EnableChannelDma and DMIC_EnableChannelFifo did not clean relevant bits.

[2.3.0]

- Improvements

    – Added new apis DMIC_ResetChannelDecimator/DMIC_EnableChannelGlobalSync/DMIC_DisableChann

[2.2.1]

- Bug Fixes

    – Fixed violations of the MISRA C-2012 rules 14.4, 17.7, 10.4, 10.3, 10.8, 14.3.

[2.2.0]

- Bug Fixes

    – Corrected the usage of feature FSL_FEATURE_DMIC_IO_HAS_NO_BYPASS.

[2.1.1]

- Improvements

    – Added feature FSL_FEATURE_DMIC_HAS_NO_IOCFG for IOCFG register.

[2.1.0]

- New Features

    – Added API DMIC_EnbleChannelInterrupt/DMIC_EnbleChannelDma to replace API DMIC_SetOperationMode.

    – Added API DMIC_SetIOCFG and marked DMIC_ConfigIO as deprecated.

    – Added API DMIC_EnableChannelSignExtend to support sign extend feature.

[2.0.5]

- Improvements

    – Changed some parameters' value of DMIC_FifoChannel API, such as enable, resetn, and trig_level. This is not possible for the current code logic, so it improves the DMIC_FifoChannel logic and fixes incorrect math logic.

[2.0.4]

- Bug Fixes

    – Fixed the issue that DMIC DMA driver(ver2.0.3) did not support calling DMIC_TransferReceiveDMA in DMA callback as it did before version 2.0.3. But calling DMIC_TransferReceiveDMA in callback is not recommended.

[2.0.3]

- New Features
- Supported linked transfer in DMIC DMA driver.
- Added new API DMIC_EnableChannelFifo/DMIC_DoFifoReset/DMIC_InstallDMADescriptor.

[2.0.2]

- New Features
  - Supported more channels in driver.

[2.0.1]

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

[2.0.0]

- Initial version.

DMIC_DMA

[2.4.2]

- Bug Fixes
  - Fixed coverity High Impact finding

[2.4.1]

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

[2.4.0]

- Bug Fixes
  - Fixed the issue that DMIC_TransferAbortReceiveDMA can not disable dmic and dma request issue.

[2.3.1]

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

[2.3.0]

- Refer DMIC driver change log 2.0.1 to 2.3.0

EMC

[2.0.4]

- Bug Fixes

    - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.8, 11.9, 14.2, 14.3, 14.4.

[2.0.3]

- Improvements

    - Used SDK_DelayAtLeastUs instead of for loop during the dynamic memory initialization.

[2.0.3]

- Improvements

    - Replaced deprecated enumerator CLOCK_GetFreq(kCLOCK_EMC) with CLOCK_GetEmcClkFreq().

[2.0.2]

- New Features

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

[2.0.1]

- Improvements

    - Added const for two BASE values.

[2.0.0]

- Initial version.

LPC_ENET

[2.3.5]

- Bug Fixes

    - Fixed ENET_GetMacAddr address byte order not matching ENET_SetMacAddr.

[2.3.4]

- Bug Fixes

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

    - Fixed the issue that ENET_DropFrame checks the buffer descriptor flag after it has been re-initialized.

    - Fixed the ENET_GetRxFrame FCS calculation issue.

– Fixed the issue that there's no valid error type in the return structure when Rx error bit is set.

[2.3.3]

- Bug Fixes

    – Fixed the issue that ENET_SetSMI uses wrong clock source to calculate the divisor.

[2.3.2]

- New features

    – Added hardware checksum acceleration support.

- Bug Fixes

    – Fixed the issue that enable/disable interrupt APIs miss part of configuration.

[2.3.1]

- Improvements

    – update ENET_SetSYSControl to support mcx family.

[2.3.0]

- Improvements

    – Added MDIO access wrapper APIs for ease of use.

[2.2.0]

- Bug Fixes

    – Corrected the timestamp retrieving code in ReadFrame.

- New Features

    – Supported zero copy Rx with new APIs.

- Improvements

    – Removed 4 bytes CRC data in ReadFrame function, not give them to user.

    – Deleted previous timestamp rings which store Tx/Rx timestamp temporarily for further retrieving. Now get Rx timestamp directly with receiving frame API, and get Tx timestamp in Tx over interrupt handler callback.

    – Added channel parameter for the SendFrame function, let user to decide which kind of frame can be sent from specified channel.

    – Supported scattered Tx buffers and more Tx configurations in SendFrame which aren't integrated.

    – Adjusted the callback location in Tx reclaim function. When use multiple BDs for Tx, only last BD transmit over interrupt event calls the callback. It simplifies the usage of Tx reclaiming.

    – Added interrupt configuration in config parameter for ENET_Init() to simplify the interrupt enable.

    – Changed the Tx/Rx descriptor name to common name rather than previous read format name which make user confused when driver uses it as write-back format.

[2.1.5]

- Bug Fixes

    - Fixed violations of the MISRA C-2012 rules 3.1,5.8,8.4,8.6,10.1,10.3,10.4,10.6,10.8,11.6,11.9,12.2,14.4,15.6,

[2.1.4]

- Bug Fixes

    - Fixed the MDC clock divider setting issue occurring when core clock range exceeds 150M.

[2.1.3]

- In ENET_StartRxTx, updated to enable TX and RX at the same time to avoid issue where ENET module could not work under 10 M.

- Changed to use CLOCK_GetCoreSysClkFreq() instead of SystemCoreClock to get accurate core clock.

[2.1.2]

- Bug Fixes

    - Fixed ENET receive issue where it sometimes lost some unicast packets. The issue is caused by the program timing issue for writing MAC_ADDR_LOW and MAC_ADDR_HIGH.

[2.1.1]

- New Features

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

[2.1.0]

- New Features

    - Added two APIs to set the ENET to ACCPET or reject the multicast frames.

[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

[2.3.4]

- Bug Fixes

  – Added internal reset of master function after arbitration lost in I2C_MasterTransferHandleIRQ(), otherwise the STAT[MSTARBLOSS] is not set in next transfer.

[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 vise 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 excution 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) hardcode.
    - 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

[2.0.10]

- Bug Fixes

    – Fixed CERT-C violations.

[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

[2.2.1]

- Improvements

    – Added macros IOCON_INV_DI and IOCON_OPENDRAIN_DI

[2.2.0]

- Improvements
  - Removed duplicate macro defintions.
  - Renamed 'IOCON_I2C_SLEW' macro to 'IOCON_I2C_MODE' to match its companion 'IO-CON_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.

LPC_LCDC

[2.0.2]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 3.1, 10.3, 10.4, 10.6, 10.7, 10.8, 14.4, 17.7.

[2.0.1]

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

[2.0.0]

- Initial version.

MCAN

[2.4.2]

- Bug Fixes
  - Fixed MISRA issue rule-10.3, rule-10.6, rule-10.7 and rule-15.7.

[2.4.1]

- Bug Fixes

    - Fixed incorrect fifo1 status on message lost.

[2.4.0]

- Improvements

    - Add MCAN_CalculateSpecifiedTimingValues() API to get CAN bit timing parameter with user-defined settings.

    - Add MCAN_FDCalculateSpecifiedTimingValues() API to get CANFD bit timing parameter with user-defined settings.

[2.3.2]

- Bug Fixes

    - Fix MISRA C-2012 issue 10.1 and 10.4.

[2.3.1]

- Bug Fixes

    - Fixed the issue that MCAN_TransferSendNonBlocking() API can't send remote frame.

[2.3.0]

- Improvements

    - Add MCAN_SetMessageRamConfig() API to perform global message RAM configure.

    - Add MCAN_EnterInitialMode() API.

[2.2.0]

- Improvements

    - Add MCAN_SetBaudRate/MCAN_SetBaudRateFD APIs to make users easy to set CAN baud rate.

[2.1.8]

- Bug Fixes

    - Add check FIFO status code in MCAN_ReadRxFifo() to avoid read back empty frame and wrong trigger the FIFO index increase.

[2.1.7]

- Bug Fixes

    - Fixed the clear error flags issue in MCAN_TransferHandleIRQ() API.

    - Fixed the Solve Tx interrupt issue in MCAN_TransferHandleIRQ() API which may abort the unhandled transfers.

    - Remove disable global tx interrupt from MCAN_TransferAbortSend API.

[2.1.6]

- Bug Fixes

    - Fixed the issue of writing 1 in the following functions.

    - MCAN_TransmitAddRequest

    - MCAN_TransmitCancelRequest

    - MCAN_ClearRxBufferStatusFlag

[2.1.5]

- Bug Fixes

    - Fix MISRA C-2012 issue.

[2.1.4]

- Improvements

    - Updated improve timing APIs to make it can calculate the CiA recommended timing configuration.

    - Implement Transmitter Delay Compensation feature.

    - Modify the default baudRateFD value to 2M.

- Bug Fixes

    - Fixed the code error issue in MCAN_ClearStatusFlag() to avoid clear all flags.

[2.1.3]

- Bug Fixes

    - Fixed the code error issue and simplified the algorithm in improved timing APIs.

        * MCAN_CalculateImprovedTimingValues

        * MCAN_FDCalculateImprovedTimingValues

[2.1.2]

- Bug Fixes

    - Fixed the non-divisible case in improved timing APIs.

        * MCAN_CalculateImprovedTimingValues

        * MCAN_FDCalculateImprovedTimingValues

[2.1.1]

- Bug Fixes

    - MISRA C-2012 issue check.

        * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.6, rule-10.7, rule-10.8, rule-11.9, rule-14.4, rule-15.5, rule-15.6, rule-15.7, rule-17.7, rule-18.4, rule-2.2, rule-21.15, rule-5.8, rule-8.3.

        * Fixed the Coverity issue of BAD_SHIFT in MCAN.

        * Fixed the issue of Pa082 warning.

* Fixed the issue of dropping interrupt flags in handler function.

[2.1.0]

- Bug Fixes
    - Fixed Coverity issue FORWARD_NULL.
    - Fixed Clang issue.
    - Fixed legacy issue in the driver and changed default bus data baud rate for CANFD.
- Improvements
    - Implemented feature for improved timing configuration.

[2.0.3]

- Improvements
    - Used memset to initialize the structure before using.
    - Added function definition comment in c file.
    - Updated source file license to SPDX BSD_3.
    - Corrected capital mistake of Fifo and fifo.
    - Reset the MCAN module in LPC drivers after clock enable.

[2.0.2]

- Bug Fixes
    - Picked MISRA fixed in release 8 branch.
    - MISRA C 2012 fixed regarding FlexCAN and MCAN address update.
- Improvements
    - Implemented for delay/retry in MCAN driver.

[2.0.1]

- Improvements
    - LPC54608 chip did not support the FD feature, so added a feature macro for it.

[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.

OTP

[2.0.1]

- Bug Fixes
    - Fixed MISRA-C 2012 violations.

[2.0.0]

- Initial version.

PINT

[2.3.0]

- Improvements
    - Add API PINT_EnableInterruptByIndex and PINT_DisableInterruptByIndex to provide more granular interrupt control.

[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.

POWER

[2.1.0]

- New features
    - Added BOD control APIs.

[2.0.0]

- initial version.

PUF

[2.2.0]

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

[2.1.6]

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

[2.1.5]

- Use common SDK delay in puf_wait_usec().

[2.1.4]

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

[2.1.3]

- Fix MISRA C-2012 issue.

[2.1.2]

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

[2.1.1]

- Fix ARMGCC build warning .

[2.1.0]

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

[2.0.3]

- Fix MISRA C-2012 issue.

[2.0.2]

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

[2.0.1]

- Bug Fixes:
    - Fixed puf_wait_usec function optimization issue.

[2.0.0]

- Initial version.

RESET

[2.4.0]

- Improvements
    - Add RESET_ReleasePeripheralReset API.

[2.0.1]

- Update component full_name to "Reset Driver".

[2.0.0]

- initial version.

RIT

[2.1.2]

- Bug Fixes
    - Fixed CERT INT31-C violations.

[2.1.1]

- Bug Fixes
    - Fixed violations of the MISRA C-2012 rules 10.3, 11.9, 17.7.

[2.1.0]

- Bug Fixes
    - Fixed issue for wrong implementation of clearing counter API in RIT driver.

[2.0.2]

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

[2.0.1]

- Bug Fixes
    - Fixed incorrect comments of some APIs.

[2.0.0]

- Initial version.

RNG

[2.1.0]

- Renamed function RNG_GetRandomData() to RNG_GetRandomDataWord(). Added function RNG_GetRandomData() which discarding next 32 words after reading RNG register which results into better entropy, as is recommended in UM.
- API is aligned with other RNG driver, having similar functionality as other RNG/TRNG drivers.

[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 SC-TIMER_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 SC-TIMER_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.

SDIF

[2.1.0]

- Improvements
    - Removed reduntant member endianMode in sdif_config_t.
    - Added error status check in function SDIF_WaitCommandDone.
    - Fixed the read fifo data incomplete issue in interrupt non-dma mode.

[2.0.15]

- Bug Fixes
    - Cleared the interrupt status before enable the interrupt to avoid interrupt generate unexpectedly.
    - Fixed the SDIF_ReadDataPortBlocking blocking at wrong condition issue.
- Improvements
    - Enabled the functionality of timeout parameter in SDIF_SendCommand.
    - Added the error recovery while sending sync clock command timeout.

[2.0.14]

- Improvements
    - Used different status code for command and data interrupt callback.
- Bug Fixes
    - Fixed the DMA descriptor attribute field unreset when configuring the current transfer DMA descriptor issue which may cause the transfer terminate unexpected.

[2.0.13]

- Improvements
    - Disabled redundant interrupt per different transfer request.
    - Disabled interrupt and reset command/data pointer in handle when transfer completes.
- Bug Fixes
    - Fixed the PA082 build warning.
    - Fixed violations of the MISRA C-2012 rules 14.4, 17.7, 10.4, 10.3, 10.8, 14.3, 10.1, 16.4, 15.7, 12.2, 11.3, 11.9.

[2.0.12]

- Bug Fixes

    - Fixed the issue that SDIF_ConfigClockDelay didn't reset the delay field before write.

    - Removed useless fifo reset code in transfer function.

    - Fixed the divider overflow issue in function SDIF_SetCardClock.

[2.0.11]

- Improvements

    - Added API SDIF_GetEnabledInterruptStatus/SDIF_GetEnabledDMAInterruptStatus and used in SDIF_TransferHandleIRQ.

    - Removed useless members interruptFlags/dmaInterruptFlags in the sdif_handle_t.

    - Improved SDIF_SendCommand with return success directly when timeout is 0.

    - Added timeout error check when sending update clock command in SDIF_SetCardClock.

    - Removed START_CMD status polling for normal command sending in SDIF_TransferBloking/SDIF_TransferNonBlocking.

    - Disabled timeout parameter in function SDIF_SendCommand.

- Bug Fixes

    - Added delay cycle for the default speed mode(400 K and 25 M) to fix the timing issue when different AHB clocks are configured.

[2.0.10]

- Bug Fixes

    - Fixed the issue that API SDIF_EnableCardClock could not clear the clock enable bit.

[2.0.9]

- Bug Fixes

    - Fixed MDK 66-D warning.

[2.0.8]

- New Features

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

    - Disabled useless interrupt while DMA is used.

    - Updated SDIF driver for one instance support two cards.

[2.0.7]

- Bug Fixes

    - Enlarged the timeout value to avoid a command conflict issue.

[2.0.6]

- Bug Fixes

    – Removed assert(srcClock_Hz <= FSL_FEATURE_SDIF_MAX_SOURCE_CLOCK).

    – Used hardware reset instead of software reset during initialization.

[2.0.5]

- New Features

    – Added non-word aligned data address and DMA descriptor address transfer support. Once one of the above addresses is not aligned, switch to host transfer mode.

- Bug Fixes

    – Fixed the issue that DMA suspended during initialization.

    – Removed useless memset function call.

[2.0.4]

- Improvements

    – Added cardInserted/cardRemoved callback function.

    – Added host base address/user data parameter for all call back functions.

[2.0.3]

- Improvements

    – Improved Clock Delay macro to allow the user to redefine and remove useless delay for clock below 25 MHz.

[2.0.2]

- Bug Fixes

    – Fixed the issue that the status flag could not be cleared entirely after transfer complete.

[2.0.1]

- New Features

    – Improved interrupt transfer callback.

- Bug Fixes

    – Added assert to limit the SDIF source clock below 52 MHz.

[2.0.0]

- Initial version.

SHA

[2.3.2]

- Add -O2 optimization for GCC to sha_process_message_data_master(), because without it the function hangs under some conditions.

[2.3.1]

- Modified sha_process_message_data_master() to ensure that MEMCTRL will be written within 64 cycles of writing last word to INDATA as is mentioned in errata, even with different optimization levels.

[2.3.0]

- Modified SHA_Update to use blocking version of AHB Master mode when its available on chip. Added SHA_UpdateNonBlocking() function which uses nonblocking AHB Master mode.
- Fixed incorrect calculation of SHA when calling SHA_Update multiple times when is CPU used to load data.
- Added Reset into SHA_ClkInit and SHA_ClkDeinit function.

[2.2.2]

- Modified SHA_Finish function. While using pseudo DMA with maximum optimization, compiler optimize out condition. Which caused block in this function and did not check flag, which has been set in interrupt.

[2.2.1]

- MISRA C-2012 issue fix.

[2.2.0]

- Support MEMADDR pseudo DMA for loading input data in SHA_Update function (LPCXpresso54018 and LPCXpresso54628).

[2.1.1]

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

[2.1.0]

- Updated "sha_ldm_stm_16_words" "sha_one_block" API to match QN9090. QN9090 has no ALIAS register.
- Added "SHA_ClkInit" "SHA_ClkInit"

[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

[2.2.3]

- Bug Fixes

  – Fixed the bug SPI_MasterTransferGetCountDMA and SPI_SlaveTransferGetCountDMA returns wrong count.

[2.2.2]

- Bug Fixes
  - Fixed the bug half duplex mode can't be used if data size is larger than 1024 bytes.

[2.2.1]

- Bug Fixes
  - Fixed MISRA 2012 11.6 issue..

[2.2.0]

- Improvements
  - Supported dataSize larger than 1024 data transmit.

SPI Flash Interface

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.

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 US-ART_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 allewed 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 occurrs 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 occurrs before TX interrupt is enabled and after txData-Size is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.

[2.1.1]

- Improvements

    – Added check for transmitter idle in USART_TransferHandleIRQ and US-ART_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 acurate 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 FIFOIN-TENSET 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

[2.1.10]

- Bug Fixes
    - Chek WWDT_RSTS instead of FSL_FEATURE_WWDT_HAS_NO_RESET to determine whether the peripheral can be reset.

[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 WWWDT 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.

LPC54S018

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 AWS IoT

aws_iot

1.7.3 FreeRTOS

FreeRTOS

1.7.4 lwIP

lwIP

1.7.5 File systemFatfs

FatFs

Chapter 2

LPC54S018

2.1 AES: AES encryption decryption driver

status_t AES_SetKey(AES_Type *base, const uint8_t *key, size_t keySize)

Sets AES key.

Sets AES key.

Parameters

- base – AES peripheral base address
- key – Input key to use for encryption or decryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from Set Key operation

status_t AES_EncryptEcb(AES_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, size_t size)

Encrypts AES using the ECB block mode.

Encrypts AES using the ECB block mode.

Parameters

- base – AES peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

Returns

Status from encrypt operation

status_t AES_DecryptEcb(AES_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, size_t size)

Decrypts AES using the ECB block mode.

Decrypts AES using the ECB block mode.

Parameters

- base – AES peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plain text

- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

Returns

Status from decrypt operation

status_t AES_EncryptCbc(AES_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])

Encrypts AES using CBC block mode.

Parameters

- base – AES peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

status_t AES_DecryptCbc(AES_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])

Decrypts AES using CBC block mode.

Parameters

- base – AES peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.

Returns

Status from decrypt operation

status_t AES_EncryptCfb(AES_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])

Encrypts AES using CFB block mode.

Parameters

- base – AES peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input Initial vector to be used as the first input block.

Returns

Status from encrypt operation

status_t AES_DecryptCfb(AES_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])

Decrypts AES using CFB block mode.

Parameters

- base – AES peripheral base address
- ciphertext – Input cipher text to decrypt

- plaintext – **[out]** Output plain text

- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

- iv – Input Initial vector to be used as the first input block.

Returns

Status from decrypt operation

status_t AES_EncryptOfb(AES_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])

Encrypts AES using OFB block mode.

Parameters

- base – AES peripheral base address

- plaintext – Input plain text to encrypt

- ciphertext – **[out]** Output cipher text

- size – Size of input and output data in bytes.

- iv – Input Initial vector to be used as the first input block.

Returns

Status from encrypt operation

status_t AES_DecryptOfb(AES_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])

Decrypts AES using OFB block mode.

Parameters

- base – AES peripheral base address

- ciphertext – Input cipher text to decrypt

- plaintext – **[out]** Output plain text

- size – Size of input and output data in bytes.

- iv – Input Initial vector to be used as the first input block.

Returns

Status from decrypt operation

status_t AES_CryptCtr(AES_Type *base, const uint8_t *input, uint8_t *output, size_t size, uint8_t counter[16], uint8_t counterlast[16], size_t *szLeft)

Encrypts or decrypts AES using CTR block mode.

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

- base – AES peripheral base address

- input – Input data for CTR block mode

- output – **[out]** Output data for CTR block mode

- size – Size of input and output data in bytes

- counter – **[inout]** Input counter (updates on return)

- counterlast – **[out]** Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.

- szLeft – **[out]** Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns

Status from crypt operation

status_t AES_EncryptTagGcm(AES_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t *iv, size_t ivSize, const uint8_t *aad, size_t aadSize, uint8_t *tag, size_t tagSize)

Encrypts AES and tags using GCM block mode.

Encrypts AES and optionally tags using GCM block mode. If plaintext is NULL, only the GHASH is calculated and output in the 'tag' field.

Parameters

- base – AES peripheral base address

- plaintext – Input plain text to encrypt

- ciphertext – **[out]** Output cipher text.

- size – Size of input and output data in bytes

- iv – Input initial vector

- ivSize – Size of the IV

- aad – Input additional authentication data

- aadSize – Input size in bytes of AAD

- tag – **[out]** Output hash tag. Set to NULL to skip tag processing.

- tagSize – Input size of the tag to generate, in bytes. Must be 4,8,12,13,14,15 or 16.

Returns

Status from encrypt operation

status_t AES_DecryptTagGcm(AES_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t *iv, size_t ivSize, const uint8_t *aad, size_t aadSize, const uint8_t *tag, size_t tagSize)

Decrypts AES and authenticates using GCM block mode.

Decrypts AES and optionally authenticates using GCM block mode. If ciphertext is NULL, only the GHASH is calculated and compared with the received GHASH in 'tag' field.

Parameters

- base – AES peripheral base address

- ciphertext – Input cipher text to decrypt

- plaintext – **[out]** Output plain text.

- size – Size of input and output data in bytes

- iv – Input initial vector

- ivSize – Size of the IV

- aad – Input additional authentication data

- aadSize – Input size in bytes of AAD

- tag – Input hash tag to compare. Set to NULL to skip tag processing.

- tagSize – Input size of the tag, in bytes. Must be 4, 8, 12, 13, 14, 15, or 16.

Returns

Status from decrypt operation

AES_BLOCK_SIZE
   AES block size in bytes

AES_IV_SIZE
   AES Input Vector size in bytes

FSL_AES_DRIVER_VERSION
   Defines LPC AES driver version 2.0.3.

   Change log:

   • Version 2.0.3

      – Edit aes_one_block() function to be interrupt safe.

   • Version 2.0.2

      – Fix MISRA-2012 issues

   • Version 2.0.1

      – GCM constant time tag comparison

   • Version 2.0.0

      – initial version

2.2 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_Sram1
      Clock gate name: Sram1.

   enumerator kCLOCK_Sram2
      Clock gate name: Sram2.

   enumerator kCLOCK_Sram3
      Clock gate name: Sram3.

   enumerator kCLOCK_Spifi
      Clock gate name: Spifi.

   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_Gpio2
    Clock gate name: Gpio2.

enumerator kCLOCK_Gpio3
    Clock gate name: Gpio3.

enumerator kCLOCK_Pint
    Clock gate name: Pint.

enumerator kCLOCK_Gint
    Clock gate name: Gint.

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_Rit
    Clock gate name: Rit.

enumerator kCLOCK_Sct0
    Clock gate name: Sct0.

enumerator kCLOCK_Mcan0
    Clock gate name: Mcan0.

enumerator kCLOCK_Mcan1
    Clock gate name: Mcan1.

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_DMic
    Clock gate name: DMic.

enumerator kCLOCK_Ct32b2
    Clock gate name: Ct32b2.

enumerator kCLOCK_Usbd0
    Clock gate name: Usbd0.

enumerator kCLOCK_Ct32b0
    Clock gate name: Ct32b0.

enumerator kCLOCK_Ct32b1
    Clock gate name: Ct32b1.

enumerator kCLOCK_BodyBias0
    Clock gate name: BodyBias0.

enumerator kCLOCK_EzhArchB0
    Clock gate name: EzhArchB0.

enumerator kCLOCK_Lcd
    Clock gate name: Lcd.

enumerator kCLOCK_Sdio
 Clock gate name: Sdio.

enumerator kCLOCK_Usbh1
 Clock gate name: Usbh1.

enumerator kCLOCK_Usbd1
 Clock gate name: Usbd1.

enumerator kCLOCK_UsbRam1
 Clock gate name: UsbRam1.

enumerator kCLOCK_Emc
 Clock gate name: Emc.

enumerator kCLOCK_Eth
 Clock gate name: Eth.

enumerator kCLOCK_Gpio4
 Clock gate name: Gpio4.

enumerator kCLOCK_Gpio5
 Clock gate name: Gpio5.

enumerator kCLOCK_Aes
 Clock gate name: Aes.

enumerator kCLOCK_Otp
 Clock gate name: Otp.

enumerator kCLOCK_Rng
 Clock gate name: Rng.

enumerator kCLOCK_FlexComm8
 Clock gate name: FlexComm8.

enumerator kCLOCK_FlexComm9
 Clock gate name: FlexComm9.

enumerator kCLOCK_MinUart8
 Clock gate name: MinUart8.

enumerator kCLOCK_MinUart9
 Clock gate name: MinUart9.

enumerator kCLOCK_LSpi8
 Clock gate name: LSpi8.

enumerator kCLOCK_LSpi9
 Clock gate name: LSpi9.

enumerator kCLOCK_BI2c8
 Clock gate name: BI2c8.

enumerator kCLOCK_BI2c9
 Clock gate name: BI2c9.

enumerator kCLOCK_FlexI2s8
 Clock gate name: FlexI2s8.

enumerator kCLOCK_FlexI2s9
 Clock gate name: FlexI2s9.

enumerator kCLOCK_Usbhmr0
    Clock gate name: Usbhmr0.

enumerator kCLOCK_Usbhsl0
    Clock gate name: Usbhsl0.

enumerator kCLOCK_Sha0
    Clock gate name: Sha0.

enumerator kCLOCK_SmartCard0
    Clock gate name: SmartCard0.

enumerator kCLOCK_SmartCard1
    Clock gate name: SmartCard1.

enumerator kCLOCK_FlexComm10
    Clock gate name: FlexComm10.

enumerator kCLOCK_Puf
    Clock gate name: Puf.

enumerator kCLOCK_Ct32b3
    Clock gate name: Ct32b3.

enumerator kCLOCK_Ct32b4
    Clock gate name: Ct32b4.

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_UsbPll
        USB1 PLL

    enumerator kCLOCK_Mclk
        MCLK

    enumerator kCLOCK_Fro12M
        FRO12M

    enumerator kCLOCK_ExtClk
        External Clock

    enumerator kCLOCK_PllOut
        PLL Output

    enumerator kCLOCK_UsbClk
        USB input

enumerator kCLOCK_WdtOsc
    Watchdog Oscillator

enumerator kCLOCK_Frg
    Frg Clock

enumerator kCLOCK_AsyncApbClk
    Async APB 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

    enumerator kCLOCK_AsyncAudioPllClk
        Async Audio PLL clock.

    enumerator kCLOCK_AsyncI2cClkFc6
        Async I2C clock.

enum __clock_attach_id
    The enumerator of clock attach Id.

    Values:

    enumerator kSYSTICK_DIV_CLK_to_SYSTICKCLK
        Attach SYSTICK_DIV_CLK to SYSTICKCLK.

    enumerator kWDT_OSC_to_SYSTICKCLK
        Attach WDT_OSC to SYSTICKCLK.

    enumerator kOSC32K_to_SYSTICKCLK
        Attach OSC32K to SYSTICKCLK.

    enumerator kFRO12M_to_SYSTICKCLK
        Attach FRO12M to SYSTICKCLK.

    enumerator kNONE_to_SYSTICKCLK
        Attach NONE to SYSTICKCLK.

    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 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 kUSB_PLL_to_CLKOUT
    Attach USB_PLL to CLKOUT.

enumerator kAUDIO_PLL_to_CLKOUT
    Attach AUDIO_PLL to CLKOUT.

enumerator kOSC32K_OSC_to_CLKOUT
    Attach OSC32K_OSC to CLKOUT.

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 kFRO12M_to_AUDIO_PLL
    Attach FRO12M to AUDIO_PLL.

enumerator kEXT_CLK_to_AUDIO_PLL
    Attach EXT_CLK to AUDIO_PLL.

enumerator kNONE_to_AUDIO_PLL
    Attach NONE to AUDIO_PLL.

enumerator kMAIN_CLK_to_SPIFI_CLK
    Attach MAIN_CLK to SPIFI_CLK.

enumerator kSYS_PLL_to_SPIFI_CLK
    Attach SYS_PLL to SPIFI_CLK.

enumerator kUSB_PLL_to_SPIFI_CLK
    Attach USB_PLL to SPIFI_CLK.

enumerator kFRO_HF_to_SPIFI_CLK
    Attach FRO_HF to SPIFI_CLK.

enumerator kAUDIO_PLL_to_SPIFI_CLK
    Attach AUDIO_PLL to SPIFI_CLK.

enumerator kNONE_to_SPIFI_CLK
    Attach NONE to SPIFI_CLK.

enumerator kFRO_HF_to_ADC_CLK
    Attach FRO_HF to ADC_CLK.

enumerator kSYS_PLL_to_ADC_CLK
    Attach SYS_PLL to ADC_CLK.

enumerator kUSB_PLL_to_ADC_CLK
    Attach USB_PLL to ADC_CLK.

enumerator kAUDIO_PLL_to_ADC_CLK
    Attach AUDIO_PLL to ADC_CLK.

enumerator kNONE_to_ADC_CLK
    Attach NONE to ADC_CLK.

enumerator kFRO_HF_to_USB0_CLK
    Attach FRO_HF to USB0_CLK.

enumerator kSYS_PLL_to_USB0_CLK
    Attach SYS_PLL to USB0_CLK.

enumerator kUSB_PLL_to_USB0_CLK
    Attach USB_PLL to USB0_CLK.

enumerator kNONE_to_USB0_CLK
    Attach NONE to USB0_CLK.

enumerator kFRO_HF_to_USB1_CLK
    Attach FRO_HF to USB1_CLK.

enumerator kSYS_PLL_to_USB1_CLK
    Attach SYS_PLL to USB1_CLK.

enumerator kUSB_PLL_to_USB1_CLK
    Attach USB_PLL to USB1_CLK.

enumerator kNONE_to_USB1_CLK
    Attach NONE to USB1_CLK.

enumerator kFRO12M_to_FLEXCOMM0
    Attach FRO12M to FLEXCOMM0.

enumerator kFRO_HF_to_FLEXCOMM0
    Attach FRO_HF to FLEXCOMM0.

enumerator kAUDIO_PLL_to_FLEXCOMM0
    Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM1
> Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM2
> Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM3
> Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM4
> Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM5
    Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM6
    Attach AUDIO_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 kAUDIO_PLL_to_FLEXCOMM7
    Attach AUDIO_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 kFRO12M_to_FLEXCOMM8
    Attach FRO12M to FLEXCOMM8.

enumerator kFRO_HF_to_FLEXCOMM8
> Attach FRO_HF to FLEXCOMM8.

enumerator kAUDIO_PLL_to_FLEXCOMM8
> Attach AUDIO_PLL to FLEXCOMM8.

enumerator kMCLK_to_FLEXCOMM8
> Attach MCLK to FLEXCOMM8.

enumerator kFRG_to_FLEXCOMM8
> Attach FRG to FLEXCOMM8.

enumerator kNONE_to_FLEXCOMM8
> Attach NONE to FLEXCOMM8.

enumerator kFRO12M_to_FLEXCOMM9
> Attach FRO12M to FLEXCOMM9.

enumerator kFRO_HF_to_FLEXCOMM9
> Attach FRO_HF to FLEXCOMM9.

enumerator kAUDIO_PLL_to_FLEXCOMM9
> Attach AUDIO_PLL to FLEXCOMM9.

enumerator kMCLK_to_FLEXCOMM9
> Attach MCLK to FLEXCOMM9.

enumerator kFRG_to_FLEXCOMM9
> Attach FRG to FLEXCOMM9.

enumerator kNONE_to_FLEXCOMM9
> Attach NONE to FLEXCOMM9.

enumerator kMAIN_CLK_to_FLEXCOMM10
> Attach MAIN_CLK to FLEXCOMM10.

enumerator kSYS_PLL_to_FLEXCOMM10
> Attach SYS_PLL to FLEXCOMM10.

enumerator kUSB_PLL_to_FLEXCOMM10
> Attach USB_PLL to FLEXCOMM10.

enumerator kFRO_HF_to_FLEXCOMM10
> Attach FRO_HF to FLEXCOMM10.

enumerator kAUDIO_PLL_to_FLEXCOMM10
> Attach AUDIO_PLL to FLEXCOMM10.

enumerator kNONE_to_FLEXCOMM10
> Attach NONE to FLEXCOMM10.

enumerator kFRO_HF_to_MCLK
> Attach FRO_HF to MCLK.

enumerator kAUDIO_PLL_to_MCLK
> Attach AUDIO_PLL to MCLK.

enumerator kNONE_to_MCLK
> Attach NONE to MCLK.

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 kFRO12M_to_DMIC
    Attach FRO12M to DMIC.

enumerator kFRO_HF_DIV_to_DMIC
    Attach FRO_HF_DIV to DMIC.

enumerator kAUDIO_PLL_to_DMIC
    Attach AUDIO_PLL to DMIC.

enumerator kMCLK_to_DMIC
    Attach MCLK to DMIC.

enumerator kMAIN_CLK_to_DMIC
    Attach MAIN_CLK to DMIC.

enumerator kWDT_OSC_to_DMIC
    Attach WDT_OSC to DMIC.

enumerator kNONE_to_DMIC
    Attach NONE to DMIC.

enumerator kMAIN_CLK_to_SCT_CLK
    Attach MAIN_CLK to SCT_CLK.

enumerator kSYS_PLL_to_SCT_CLK
    Attach SYS_PLL to SCT_CLK.

enumerator kFRO_HF_to_SCT_CLK
    Attach FRO_HF to SCT_CLK.

enumerator kAUDIO_PLL_to_SCT_CLK
    Attach AUDIO_PLL to SCT_CLK.

enumerator kNONE_to_SCT_CLK
    Attach NONE to SCT_CLK.

enumerator kMAIN_CLK_to_LCD_CLK
    Attach MAIN_CLK to LCD_CLK.

enumerator kLCDCLKIN_to_LCD_CLK
    Attach LCDCLKIN to LCD_CLK.

enumerator kFRO_HF_to_LCD_CLK
    Attach FRO_HF to LCD_CLK.

enumerator kNONE_to_LCD_CLK
    Attach NONE to LCD_CLK.

enumerator kMAIN_CLK_to_SDIO_CLK
    Attach MAIN_CLK to SDIO_CLK.

enumerator kSYS_PLL_to_SDIO_CLK
    Attach SYS_PLL to SDIO_CLK.

enumerator kUSB_PLL_to_SDIO_CLK
    Attach USB_PLL to SDIO_CLK.

enumerator kFRO_HF_to_SDIO_CLK
    Attach FRO_HF to SDIO_CLK.

enumerator kAUDIO_PLL_to_SDIO_CLK
    Attach AUDIO_PLL to SDIO_CLK.

enumerator kNONE_to_SDIO_CLK
    Attach NONE to SDIO_CLK.

enumerator kMAIN_CLK_to_ASYNC_APB
    Attach MAIN_CLK to ASYNC_APB.

enumerator kFRO12M_to_ASYNC_APB
    Attach FRO12M to ASYNC_APB.

enumerator kAUDIO_PLL_to_ASYNC_APB
    Attach AUDIO_PLL to ASYNC_APB.

enumerator kI2C_CLK_FC6_to_ASYNC_APB
    Attach I2C_CLK_FC6 to ASYNC_APB.

enumerator kNONE_to_NONE
    Attach NONE to NONE.

enum __clock_div_name
    Clock dividers.

    Values:

    enumerator kCLOCK_DivSystickClk
        Systick Clock Divider.

    enumerator kCLOCK_DivArmTrClkDiv
        Arm Tr Clk Div Divider.

    enumerator kCLOCK_DivCan0Clk
        Can0 Clock Divider.

    enumerator kCLOCK_DivCan1Clk
        Can1 Clock Divider.

    enumerator kCLOCK_DivSmartCard0Clk
        Smart Card0 Clock Divider.

    enumerator kCLOCK_DivSmartCard1Clk
        Smart Card1 Clock Divider.

    enumerator kCLOCK_DivAhbClk
        Ahb Clock Divider.

    enumerator kCLOCK_DivClkOut
        Clk Out Divider.

    enumerator kCLOCK_DivFrohfClk
        Frohf Clock Divider.

enumerator kCLOCK_DivSpifiClk
    Spifi Clock Divider.

enumerator kCLOCK_DivAdcAsyncClk
    Adc Async Clock Divider.

enumerator kCLOCK_DivUsb0Clk
    Usb0 Clock Divider.

enumerator kCLOCK_DivUsb1Clk
    Usb1 Clock Divider.

enumerator kCLOCK_DivFrg
    Frg Divider.

enumerator kCLOCK_DivDmicClk
    Dmic Clock Divider.

enumerator kCLOCK_DivMClk
    I2S MCLK Clock Divider.

enumerator kCLOCK_DivLcdClk
    Lcd Clock Divider.

enumerator kCLOCK_DivSctClk
    Sct Clock Divider.

enumerator kCLOCK_DivEmcClk
    Emc Clock Divider.

enumerator kCLOCK_DivSdioClk
    Sdio clock divider.

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

    enumerator kStatus_PLL_CCOTooLow
        Requested CCO rate isn't possible

    enumerator kStatus_PLL_CCOTooHigh
        Requested CCO 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_UsbSrcUsbPll
>>   Use USB PLL clock.
>
>   enumerator kCLOCK_UsbSrcNone
>>   Use None, this may be selected in order to reduce power when no output is needed.

enum __usb_pll_psel
>   USB PDEL Divider.
>
>   *Values:*
>
>   enumerator pSel_Divide_1
>
>   enumerator pSel_Divide_2
>
>   enumerator pSel_Divide_4
>
>   enumerator pSel_Divide_8

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 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.

typedef enum *_usb_pll_psel* usb__pll__psel

USB PDEL Divider.

typedef struct *_usb_pll_setup* usb__pll__setup__t

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

static inline void CLOCK__EnableClock(*clock_ip_name_t* clk)

static inline void CLOCK__DisableClock(*clock_ip_name_t* clk)

void CLOCK__SetupFROClocking(uint32_t froFreq)

Initialize the Core clock to given frequency (12, 48 or 96 MHz), this API is implememnt in ROM code. Turns on FRO and uses default CCO, if freq is 12000000, then high speed output is off, else high speed output is enabled. Usage: CLOCK_SetupFROClocking(frequency), (frequency must be one of 12, 48 or 96 MHz) Note: Need to make sure ROM and OTP has power(PDRUNCFG0[17,29]= 0U) before calling this API since this API is implemented in ROM code and the FROHF TRIM value is stored in OTP.

Parameters

- froFreq – target fro frequency.

Returns

Nothing

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 fuction 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

uint32_t CLOCK__GetFreq(*clock_name_t* clockName)

>   Return Frequency of selected clock.

>>   **Returns**

>>>   Frequency of selected clock

uint32_t CLOCK__GetFro12MFreq(**void**)

>   Return Frequency of FRO 12MHz.

>>   **Returns**

>>>   Frequency of FRO 12MHz

uint32_t CLOCK__GetClockOutClkFreq(**void**)

>   Return Frequency of ClockOut.

>>   **Returns**

>>>   Frequency of ClockOut

uint32_t CLOCK__GetSpifiClkFreq(**void**)

>   Return Frequency of Spifi Clock.

>>   **Returns**

>>>   Frequency of Spifi.

uint32_t CLOCK__GetAdcClkFreq(**void**)

>   Return Frequency of Adc Clock.

>>   **Returns**

>>>   Frequency of Adc Clock.

uint32_t CLOCK__GetMCanClkFreq(**uint32_t** MCanSel)

>   brief Return Frequency of MCAN Clock param MCanSel : 0U: MCAN0; 1U: MCAN1 return Frequency of MCAN Clock

uint32_t CLOCK__GetUsb0ClkFreq(**void**)

>   Return Frequency of Usb0 Clock.

>>   **Returns**

>>>   Frequency of Usb0 Clock.

uint32_t CLOCK__GetUsb1ClkFreq(**void**)

>   Return Frequency of Usb1 Clock.

>>   **Returns**

>>>   Frequency of Usb1 Clock.

uint32_t CLOCK__GetMclkClkFreq(**void**)

>   Return Frequency of MClk Clock.

>>   **Returns**

>>>   Frequency of MClk Clock.

uint32_t CLOCK__GetSctClkFreq(**void**)

>   Return Frequency of SCTimer Clock.

>>   **Returns**

>>>   Frequency of SCTimer Clock.

uint32_t CLOCK__GetSdioClkFreq(**void**)

>   Return Frequency of SDIO Clock.

>>   **Returns**

>>>   Frequency of SDIO Clock.

uint32_t CLOCK_GetLcdClkFreq(**void**)

 Return Frequency of LCD Clock.

   Returns

    Frequency of LCD Clock.

uint32_t CLOCK_GetLcdClkIn(**void**)

 Return Frequency of LCD CLKIN Clock.

   Returns

    Frequency of LCD CLKIN Clock.

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_GetFrgClkFreq(**void**)

 Return Frequency of frg.

   Returns

    Frequency of FRG

uint32_t CLOCK_GetDmicClkFreq(**void**)

 Return Frequency of dmic.

   Returns

    Frequency of DMIC

uint32_t CLOCK_SetFRGClock(**uint32_t freq**)

 Set FRG Clk.

   Returns

    1: if set FRG CLK successfully. 0: if set FRG CLK fail.

uint32_t CLOCK_GetPllOutFreq(**void**)

 Return Frequency of PLL.

   Returns

    Frequency of PLL

uint32_t CLOCK_GetUsbPllOutFreq(**void**)

 Return Frequency of USB PLL.

   Returns

    Frequency of PLL

uint32_t CLOCK_GetAudioPllOutFreq(**void**)

 Return Frequency of AUDIO 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_GetFRGInputClock(**void**)

>   return FRG Clk

>   > **Returns**
>   >   Frequency of FRG CLK.

___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

___STATIC_INLINE uint32_t CLOCK_GetEmcClkFreq (void)

>   Return EMC source.

>   > **Returns**
>   >   EMC source

uint32_t CLOCK_GetAudioPLLInClockRate(**void**)

>   Return Audio PLL input clock rate.

>   > **Returns**
>   >   Audio PLL input clock rate

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

uint32_t CLOCK_GetAudioPLLOutClockRate(bool recompute)

Return System AUDIO PLL output clock rate.

Note: The AUDIO 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 AUDIO PLL rate recomputation if true

Returns

System AUDIO PLL output clock rate

uint32_t CLOCK_GetUsbPLLOutClockRate(bool recompute)

Return System USB PLL output clock rate.

Note: The USB 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 USB PLL rate recomputation if true

Returns

System USB 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

___STATIC_INLINE bool CLOCK_IsUsbPLLLocked (void)

Check if USB PLL is locked or not.

Returns

true if the USB PLL is locked, false if not locked

__STATIC_INLINE bool CLOCK_IsAudioPLLLocked (void)

> Check if AUDIO PLL is locked or not.

> > **Returns**
> > true if the AUDIO PLL is locked, false if not locked

__STATIC_INLINE void CLOCK_Enable_SysOsc (bool enable)

> Enables and disables SYS OSC.

> enable : true to enable SYS OSC, false to disable SYS OSC

void CLOCK_SetStoredPLLClockRate(uint32_t rate)

> Store the current PLL rate.

> > **Parameters**

> > > • rate – Current rate of the PLL

> > **Returns**
> > Nothing

void CLOCK_SetStoredAudioPLLClockRate(uint32_t rate)

> Store the current AUDIO 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 the setup structure will generate

uint32_t CLOCK_GetAudioPLLOutFromSetup(*pll_setup_t* *pSetup)

> Return System AUDIO PLL output clock rate from setup structure.

> > **Parameters**

> > > • pSetup – : Pointer to a PLL setup structure

> > **Returns**
> > System PLL output clock rate the setup structure will generate

uint32_t CLOCK_GetAudioPLLOutFromFractSetup(*pll_setup_t* *pSetup)

> Return System AUDIO PLL output clock rate from audio fractioanl setup structure.

> > **Parameters**

> > > • pSetup – : Pointer to a PLL setup structure

> > **Returns**
> > System PLL output clock rate the setup structure will generate

uint32_t CLOCK_GetUsbPLLOutFromSetup(const *usb_pll_setup_t* *pSetup)

> Return System USB PLL output clock rate from setup structure.

> > **Parameters**

> > > • pSetup – : Pointer to a PLL setup structure

> > **Returns**
> > System PLL output clock rate the setup structure will generate

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_SetupAudioPLLData(*pll_config_t* *pControl, *pll_setup_t* *pSetup)

　　Set AUDIO PLL output based on the passed AUDIO 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 systen 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_SetupAudioPLLPrec(*pll_setup_t* *pSetup, uint32_t flagcfg)

　　Set AUDIO PLL output from AUDIOPLL 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 AUDIO PLL, wait for PLL lock, and adjust system voltages to the new AUDIOPLL rate. The function will not alter any source clocks (ie, main systen clock) that may use the AUDIO 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__SetupAudioPLLPrecFract(*pll_setup_t* *pSetup, uint32_t flagcfg)

Set AUDIO PLL output from AUDIOPLL setup structure using the Audio Fractional divider register(precise

frequency)

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

pll_error_t CLOCK__SetAudioPLLFreq(const *pll_setup_t* *pSetup)

Set Audio PLL output from Audio PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the Audio PLL with the new setup data, and then optionally powerup the PLL, wait for Audio PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main systen clock) that may use the Audio 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 Audio PLL setup error code

pll_error_t CLOCK_SetUsbPLLFreq(const *usb_pll_setup_t* *pSetup)

 Set USB PLL output from USB PLL setup structure (precise frequency)

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

  Parameters

    • pSetup – : Pointer to populated USB PLL setup structure

  Returns

    kStatus_PLL_Success on success, or USB 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 systen 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_DisableUsbDevicefs0Clock(*clock_ip_name_t* clk)

 Disable USB clock.

 Disable USB clock.

bool CLOCK_EnableUsbfs0DeviceClock(*clock_usb_src_t* src, uint32_t freq)

 Enable USB Device FS clock.

  Parameters

    • src – : clock source

    • freq – clock frequency Enable USB Device Full Speed clock.

bool CLOCK_EnableUsbfs0HostClock(*clock_usb_src_t* src, uint32_t freq)

 Enable USB HOST FS clock.

  Parameters

    • src – : clock source

    • freq – clock frequency Enable USB HOST Full Speed clock.

void CLOCK_SetStoredUsbPLLClockRate(uint32_t rate)

 Set the current Usb PLL Rate.

bool CLOCK_EnableUsbhs0DeviceClock(*clock_usb_src_t* src, uint32_t freq)

 Enable USB Device HS clock.

  Parameters

- src – : clock source

- freq – clock frequency Enable USB Device High Speed clock.

bool CLOCK_EnableUsbhs0HostClock(*clock_usb_src_t* src, uint32_t freq)

Enable USB HOST HS clock.

Parameters

- src – : clock source

- freq – clock frequency Enable USB HOST High Speed clock.

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.3.3.

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

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.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

ADC_CLOCKS

Clock ip name array for ADC.

ROM_CLOCKS

Clock ip name array for ROM.

SRAM_CLOCKS

Clock ip name array for SRAM.

FLASH_CLOCKS

Clock ip name array for FLASH.

FMC_CLOCKS

Clock ip name array for FMC.

EEPROM_CLOCKS

Clock ip name array for EEPROM.

SPIFI_CLOCKS

Clock ip name array for SPIFI.

INPUTMUX_CLOCKS

Clock ip name array for INPUTMUX.

IOCON_CLOCKS

Clock ip name array for IOCON.

GPIO_CLOCKS

Clock ip name array for GPIO.

PINT_CLOCKS

Clock ip name array for PINT.

GINT_CLOCKS

Clock ip name array for GINT.

DMA_CLOCKS

Clock ip name array for DMA.

CRC_CLOCKS

Clock ip name array for CRC.

WWDT_CLOCKS

Clock ip name array for WWDT.

RTC_CLOCKS

Clock ip name array for RTC.

ADC0_CLOCKS

Clock ip name array for ADC0.

MRT_CLOCKS

Clock ip name array for MRT.

RIT_CLOCKS

Clock ip name array for RIT.

SCT_CLOCKS

Clock ip name array for SCT0.

MCAN_CLOCKS

Clock ip name array for MCAN.

UTICK_CLOCKS

Clock ip name array for UTICK.

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.

DMIC_CLOCKS

Clock ip name array for DMIC.

CTIMER_CLOCKS

Clock ip name array for CT32B.

LCD_CLOCKS

Clock ip name array for LCD.

SDIO_CLOCKS

Clock ip name array for SDIO.

USBRAM_CLOCKS

    Clock ip name array for USBRAM.

EMC_CLOCKS

    Clock ip name array for EMC.

ETH_CLOCKS

    Clock ip name array for ETH.

AES_CLOCKS

    Clock ip name array for AES.

OTP_CLOCKS

    Clock ip name array for OTP.

RNG_CLOCKS

    Clock ip name array for RNG.

USBHMR0_CLOCKS

    Clock ip name array for USBHMR0.

USBHSL0_CLOCKS

    Clock ip name array for USBHSL0.

SHA0_CLOCKS

    Clock ip name array for SHA0.

SMARTCARD_CLOCKS

    Clock ip name array for SMARTCARD.

USBD_CLOCKS

    Clock ip name array for USBD.

USBH_CLOCKS

    Clock ip name array for USBH.

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

AHB_CLK_CTRL2

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_OFFSET(mux)

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)

CLR_ID_ITEM_SEL(connection)

CM_STICKCLKSEL

CM_MAINCLKSELA

CM_MAINCLKSELB

CM_CLKOUTCLKSELA

CM_SYSPLLCLKSEL

CM_AUDPLLCLKSEL

CM_SPIFICLKSEL

CM_ADCASYNCCLKSEL

CM_USB0CLKSEL

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_MCLKCLKSEL

CM_FRGCLKSEL

CM_DMICCLKSEL

CM__SCTCLKSEL

CM__LCDCLKSEL

CM__SDIOCLKSEL

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 dyanmic 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 pwoered on

PLL__SETUPFLAG__ADGVOLT

Optimize system voltage for the new PLL rate

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

uint32_t pllctrl

PLL control register SYSPLLCTRL

uint32_t pllndec

PLL NDEC register SYSPLLNDEC

uint32_t pllpdec

PLL PDEC register SYSPLLPDEC

uint32_t pllmdec

PLL MDEC registers SYSPLLPDEC

uint32_t pllRate

    Acutal PLL rate

uint32_t audpllfrac

    only aduio PLL has this function

uint32_t flags

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

uint8_t msel

    USB PLL control register msel:1U-256U

uint8_t psel

    USB PLL control register psel:only support inter 1U 2U 4U 8U

uint8_t nsel

    USB PLL control register nsel:only suppoet inter 1U 2U 3U 4U

bool direct

    USB PLL CCO output control

bool bypass

    USB PLL inout clock bypass control

bool fbsel

    USB PLL ineter mode and non-integer mode control

uint32_t inputRate

    USB PLL input rate

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.

struct __usb_pll_setup

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

2.3 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.

2.4 CTIMER: Standard counter/timers

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

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

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

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

Parameters
- base – Ctimer peripheral base address

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

The default values are:

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

Parameters

- config – Pointer to the user configuration structure.

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

Configures the PWM signal parameters.

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

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

---

Parameters

- base – Ctimer peripheral base address

- pwmPeriodChannel – Specify the channel to control the PWM period

- matchChannel – Match pin to be used to output the PWM signal

- pwmPeriod – PWM period match value

- pulsePeriod – Pulse width match value

- enableInt – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

Returns

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

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

Configures the PWM signal parameters.

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

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

Parameters

- base – Ctimer peripheral base address

- pwmPeriodChannel – Specify the channel to control the PWM period

- matchChannel – Match pin to be used to output the PWM signal

- dutyCyclePercent – PWM pulse width; the value should be between 0 to 100

- pwmFreq_Hz – PWM signal frequency in Hz

- srcClock_Hz – Timer counter clock in Hz

- enableInt – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

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

Updates the pulse period of an active PWM signal.

Parameters

- base – Ctimer peripheral base address

- matchChannel – Match pin to be used to output the PWM signal

- pulsePeriod – New PWM pulse width match value

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

Updates the duty cycle of an active PWM signal.

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

Parameters

- base – Ctimer peripheral base address

- pwmPeriodChannel – Specify the channel to control the PWM period

- matchChannel – Match pin to be used to output the PWM signal

- dutyCyclePercent – New PWM pulse width; the value should be between 0 to 100

Returns

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

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

Enables the selected Timer interrupts.

Parameters

- base – Ctimer peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

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

Disables the selected Timer interrupts.

Parameters

- base – Ctimer peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

static inline uint32_t CTIMER_GetEnabledInterrupts(CTIMER_Type *base)

Gets the enabled Timer interrupts.

Parameters

> • base – Ctimer peripheral base address

> **Returns**
>> The enabled interrupts. This is the logical OR of members of the enumeration ctimer_interrupt_enable_t

static inline uint32_t CTIMER_GetStatusFlags(CTIMER_Type *base)

> Gets the Timer status flags.

> **Parameters**

>> • base – Ctimer peripheral base address

> **Returns**
>> The status flags. This is the logical OR of members of the enumeration ctimer_status_flags_t

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

> Clears the Timer status flags.

> **Parameters**

>> • base – Ctimer peripheral base address

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

static inline void CTIMER_StartTimer(CTIMER_Type *base)

> Starts the Timer counter.

> **Parameters**

>> • base – Ctimer peripheral base address

static inline void CTIMER_StopTimer(CTIMER_Type *base)

> Stops the Timer counter.

> **Parameters**

>> • base – Ctimer peripheral base address

FSL_CTIMER_DRIVER_VERSION

> Version 2.3.4

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 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

2.4. CTIMER: Standard counter/timers 171

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 reser 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

2.5 DMA: Direct Memory Access Controller Driver

void DMA_Init(DMA_Type *base)
> Initializes DMA peripheral.
>
> This function enable the DMA clock, set descriptor table and enable DMA peripheral.
>
> > **Parameters**
> > - base – DMA peripheral base address.

void DMA_Deinit(DMA_Type *base)
> Deinitializes DMA peripheral.
>
> This function gates the DMA clock.
>
> > **Parameters**
> > - base – DMA peripheral base address.

void DMA_InstallDescriptorMemory(DMA_Type *base, void *addr)
> Install DMA descriptor memory.
>
> This function used to register DMA descriptor memory for linked transfer, a typical case is ping pong transfer which will request more than one DMA descriptor memory space, althrough 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 reisters.

> **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 superceded 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.

2.5. DMA: Direct Memory Access Controller Driver 179

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

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

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

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

Parameters

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

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

Submits the DMA channel transfer request.

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

a. 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)
```

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);
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_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 2.5.4.

_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

2.5. DMA: Direct Memory Access Controller Driver

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_DESCRIPTORS(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_DESCRIPTORS_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**

2.5. DMA: Direct Memory Access Controller Driver 187

- name – Allocate decriptor name.

- number – Number of descriptor to be allocated.

DMA_ALLOCATE_LINK_DESCRIPTORS(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_DESCRIPTORS_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 algin 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

2.6 DMIC: Digital Microphone

2.7 DMIC DMA Driver

status_t DMIC_TransferCreateHandleDMA(DMIC_Type *base, *dmic_dma_handle_t* *handle, *dmic_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *rxDmaHandle)

> Initializes the DMIC handle which is used in transactional functions.

> **Parameters**

>> • base – DMIC peripheral base address.

>> • handle – Pointer to dmic_dma_handle_t structure.

>> • callback – Callback function.

>> • userData – User data.

>> • rxDmaHandle – User-requested DMA handle for RX DMA transfer.

status_t DMIC_TransferReceiveDMA(DMIC_Type *base, *dmic_dma_handle_t* *handle, *dmic_transfer_t* *xfer, uint32_t channel)

> 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 – DMIC DMA transfer structure. See dmic_transfer_t.

>> • channel – DMIC start channel number.

> **Return values**
>> kStatus_Success –

void DMIC_TransferAbortReceiveDMA(DMIC_Type *base, *dmic_dma_handle_t* *handle)

> Aborts the received data using DMA.

> This function aborts the received data using DMA.

> **Parameters**

>> • base – DMIC peripheral base address

>> • handle – Pointer to dmic_dma_handle_t structure

status_t DMIC_TransferGetReceiveCountDMA(DMIC_Type *base, *dmic_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 – DMIC peripheral base address.

>> • handle – DMIC 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 DMIC_InstallDMADescriptorMemory(*dmic_dma_handle_t* *handle, void *linkAddr, size_t linkNum)

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, it should be called after DMIC_TransferCreateHandleDMA. User should be take care about the address of DMA descriptor pool which required align with 16BYTE at least.

Parameters

- handle – Pointer to DMA channel transfer handle.

- linkAddr – DMA link descriptor address.

- linkNum – DMA link descriptor number.

FSL_DMIC_DMA_DRIVER_VERSION

DMIC DMA driver version 2.4.2.

typedef struct *_dmic_transfer* dmic_transfer_t

DMIC transfer structure.

typedef struct *_dmic_dma_handle* dmic_dma_handle_t

typedef void (*dmic_dma_transfer_callback_t)(DMIC_Type *base, *dmic_dma_handle_t* *handle, *status_t* status, void *userData)

DMIC transfer callback function.

struct __dmic_transfer

#include <fsl_dmic_dma.h> DMIC transfer structure.

Public Members

void *data

The buffer of data to be transfer.

uint8_t dataWidth

DMIC support 16bit/32bit

size_t dataSize

The byte count to be transfer.

uint8_t dataAddrInterleaveSize

destination address interleave size

struct *_dmic_transfer* *linkTransfer

use to support link transfer

struct __dmic_dma_handle

#include <fsl_dmic_dma.h> DMIC DMA handle.

Public Members

DMIC_Type *base

DMIC peripheral base address.

dma_handle_t *rxDmaHandle
    The DMA RX channel used.

dmic_dma_transfer_callback_t callback
    Callback function.

void *userData
    DMIC callback function parameter.

size_t transferSize
    Size of the data to receive.

volatile uint8_t state
    Internal state of DMIC DMA transfer

uint32_t channel
    DMIC channel used.

bool isChannelValid
    DMIC channel initialization flag

dma_descriptor_t *desLink
    descriptor pool pointer

size_t linkNum
    number of descriptor in descriptors pool

2.8 DMIC Driver

uint32_t DMIC_GetInstance(DMIC_Type *base)
    Get the DMIC instance from peripheral base address.

> **Parameters**
>
> > • base – DMIC peripheral base address.
>
> **Returns**
> > DMIC instance.

void DMIC_Init(DMIC_Type *base)
    Turns DMIC Clock on.

> **Parameters**
>
> > • base – : DMIC base
>
> **Returns**
> > Nothing

void DMIC_DeInit(DMIC_Type *base)
    Turns DMIC Clock off.

> **Parameters**
>
> > • base – : DMIC base
>
> **Returns**
> > Nothing

void DMIC_ConfigIO(DMIC_Type *base, *dmic_io_t* config)
    Configure DMIC io.

Deprecated:

> Do not use this function. It has been superceded by DMIC_SetIOCFG

> **Parameters**

>> • base – : The base address of DMIC interface

>> • config – : DMIC io configuration

> **Returns**
>> Nothing

static inline void DMIC_SetIOCFG(DMIC_Type *base, uint32_t sel)

> Stereo PDM select.

> **Parameters**

>> • base – : The base address of DMIC interface

>> • sel – : Reference dmic_io_t, can be a single or combination value of dmic_io_t.

> **Returns**
>> Nothing

void DMIC_SetOperationMode(DMIC_Type *base, *operation_mode_t* mode)

> Set DMIC operating mode.

Deprecated:

> Do not use this function. It has been superceded by DMIC_EnableChannelInterrupt, DMIC_EnableChannelDma.

> **Parameters**

>> • base – : The base address of DMIC interface

>> • mode – : DMIC mode

> **Returns**
>> Nothing

void DMIC_Use2fs(DMIC_Type *base, bool use2fs)

> Configure Clock scaling.

> **Parameters**

>> • base – : The base address of DMIC interface

>> • use2fs – : clock scaling

> **Returns**
>> Nothing

void DMIC_CfgChannelDc(DMIC_Type *base, *dmic_channel_t* channel, *dc_removal_t* dc_cut_level, uint32_t post_dc_gain_reduce, bool saturate16bit)

> Configure DMIC channel.

> **Parameters**

>> • base – : The base address of DMIC interface

>> • channel – : DMIC channel

>> • dc_cut_level – : dc_removal_t, Cut off Frequency

>> • post_dc_gain_reduce – : Fine gain adjustment in the form of a number of bits to downshift.

- saturate16bit – : If selects 16-bit saturation.

void DMIC_ConfigChannel(DMIC_Type *base, *dmic_channel_t* channel, *stereo_side_t* side, *dmic_channel_config_t* *channel_config)

Configure DMIC channel.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel
- side – : stereo_side_t, choice of left or right
- channel_config – : Channel configuration

Returns

Nothing

void DMIC_EnableChannnel(DMIC_Type *base, uint32_t channelmask)

Enable a particualr channel.

Parameters

- base – : The base address of DMIC interface
- channelmask – reference _dmic_channel_mask

Returns

Nothing

void DMIC_FifoChannel(DMIC_Type *base, uint32_t channel, uint32_t trig_level, uint32_t enable, uint32_t resetn)

Configure fifo settings for DMIC channel.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel
- trig_level – : FIFO trigger level
- enable – : FIFO level
- resetn – : FIFO reset

Returns

Nothing

static inline void DMIC_EnableChannelInterrupt(DMIC_Type *base, *dmic_channel_t* channel, bool enable)

Enable a particualr channel interrupt request.

Parameters

- base – : The base address of DMIC interface
- channel – : Channel selection
- enable – : true is enable, false is disable

static inline void DMIC_EnableChannelDma(DMIC_Type *base, *dmic_channel_t* channel, bool enable)

Enable a particualr channel dma request.

Parameters

- base – : The base address of DMIC interface
- channel – : Channel selection

- enable – : true is enable, false is disable

static inline void DMIC_EnableChannelFifo(DMIC_Type *base, *dmic_channel_t* channel, bool enable)

Enable a particualr channel fifo.

Parameters

- base – : The base address of DMIC interface

- channel – : Channel selection

- enable – : true is enable, false is disable

static inline void DMIC_DoFifoReset(DMIC_Type *base, *dmic_channel_t* channel)

Channel fifo reset.

Parameters

- base – : The base address of DMIC interface

- channel – : Channel selection

static inline uint32_t DMIC_FifoGetStatus(DMIC_Type *base, uint32_t channel)

Get FIFO status.

Parameters

- base – : The base address of DMIC interface

- channel – : DMIC channel

Returns

FIFO status

static inline void DMIC_FifoClearStatus(DMIC_Type *base, uint32_t channel, uint32_t mask)

Clear FIFO status.

Parameters

- base – : The base address of DMIC interface

- channel – : DMIC channel

- mask – : Bits to be cleared

Returns

FIFO status

static inline uint32_t DMIC_FifoGetData(DMIC_Type *base, uint32_t channel)

Get FIFO data.

Parameters

- base – : The base address of DMIC interface

- channel – : DMIC channel

Returns

FIFO data

static inline uint32_t DMIC_FifoGetAddress(DMIC_Type *base, uint32_t channel)

Get FIFO address.

Parameters

- base – : The base address of DMIC interface

- channel – : DMIC channel

Returns

FIFO data

void DMIC_EnableIntCallback(DMIC_Type *base, *dmic_callback_t* cb)

> Enable callback.

> This function enables the interrupt for the selected DMIC peripheral. The callback function is not enabled until this function is called.

> > **Parameters**
> > > - base – Base address of the DMIC peripheral.
> > > - cb – callback Pointer to store callback function.

> > **Return values**
> > > None. –

void DMIC_DisableIntCallback(DMIC_Type *base, *dmic_callback_t* cb)

> Disable callback.

> This function disables the interrupt for the selected DMIC peripheral.

> > **Parameters**
> > > - base – Base address of the DMIC peripheral.
> > > - cb – callback Pointer to store callback function..

> > **Return values**
> > > None. –

static inline void DMIC_SetGainNoiseEstHwvad(DMIC_Type *base, uint32_t value)

> Sets the gain value for the noise estimator.

> > **Parameters**
> > > - base – DMIC base pointer
> > > - value – gain value for the noise estimator.

> > **Return values**
> > > None. –

static inline void DMIC_SetGainSignalEstHwvad(DMIC_Type *base, uint32_t value)

> Sets the gain value for the signal estimator.

> > **Parameters**
> > > - base – DMIC base pointer
> > > - value – gain value for the signal estimator.

> > **Return values**
> > > None. –

static inline void DMIC_SetFilterCtrlHwvad(DMIC_Type *base, uint32_t value)

> Sets the hwvad filter cutoff frequency parameter.

> > **Parameters**
> > > - base – DMIC base pointer
> > > - value – cut off frequency value.

> > **Return values**
> > > None. –

static inline void DMIC_SetInputGainHwvad(DMIC_Type *base, uint32_t value)

> Sets the input gain of hwvad.

> > **Parameters**
> > > - base – DMIC base pointer

- value – input gain value for hwvad.

> **Return values**
> None. –

static inline void DMIC_CtrlClrIntrHwvad(DMIC_Type *base, bool st10)

> Clears hwvad internal interrupt flag.

> **Parameters**

- base – DMIC base pointer

- st10 – bit value.

> **Return values**
> None. –

static inline void DMIC_FilterResetHwvad(DMIC_Type *base, bool rstt)

> Resets hwvad filters.

> **Parameters**

- base – DMIC base pointer

- rstt – Reset bit value.

> **Return values**
> None. –

static inline uint16_t DMIC_GetNoiseEnvlpEst(DMIC_Type *base)

> Gets the value from output of the filter z7.

> **Parameters**

- base – DMIC base pointer

> **Return values**
> output – of filter z7.

void DMIC_HwvadEnableIntCallback(DMIC_Type *base, *dmic_hwvad_callback_t* vadcb)

> Enable hwvad callback.

> This function enables the hwvad interrupt for the selected DMIC peripheral. The callback function is not enabled until this function is called.

> **Parameters**

- base – Base address of the DMIC peripheral.

- vadcb – callback Pointer to store callback function.

> **Return values**
> None. –

void DMIC_HwvadDisableIntCallback(DMIC_Type *base, *dmic_hwvad_callback_t* vadcb)

> Disable callback.

> This function disables the hwvad interrupt for the selected DMIC peripheral.

> **Parameters**

- base – Base address of the DMIC peripheral.

- vadcb – callback Pointer to store callback function..

> **Return values**
> None. –

FSL_DMIC_DRIVER_VERSION

> DMIC driver version 2.3.3.

_dmic_status DMIC transfer status.

Values:

enumerator kStatus_DMIC_Busy
DMIC is busy

enumerator kStatus_DMIC_Idle
DMIC is idle

enumerator kStatus_DMIC_OverRunError
DMIC over run Error

enumerator kStatus_DMIC_UnderRunError
DMIC under run Error

enum __operation_mode
DMIC different operation modes.

Values:

enumerator kDMIC_OperationModeInterrupt
Interrupt mode

enumerator kDMIC_OperationModeDma
DMA mode

enum __stereo_side
DMIC left/right values.

Values:

enumerator kDMIC_Left
Left Stereo channel

enumerator kDMIC_Right
Right Stereo channel

enum pdm_div_t
DMIC Clock pre-divider values.

Values:

enumerator kDMIC_PdmDiv1
DMIC pre-divider set in divide by 1

enumerator kDMIC_PdmDiv2
DMIC pre-divider set in divide by 2

enumerator kDMIC_PdmDiv3
DMIC pre-divider set in divide by 3

enumerator kDMIC_PdmDiv4
DMIC pre-divider set in divide by 4

enumerator kDMIC_PdmDiv6
DMIC pre-divider set in divide by 6

enumerator kDMIC_PdmDiv8
DMIC pre-divider set in divide by 8

enumerator kDMIC_PdmDiv12
DMIC pre-divider set in divide by 12

enumerator kDMIC_PdmDiv16
    DMIC pre-divider set in divide by 16

enumerator kDMIC_PdmDiv24
    DMIC pre-divider set in divide by 24

enumerator kDMIC_PdmDiv32
    DMIC pre-divider set in divide by 32

enumerator kDMIC_PdmDiv48
    DMIC pre-divider set in divide by 48

enumerator kDMIC_PdmDiv64
    DMIC pre-divider set in divide by 64

enumerator kDMIC_PdmDiv96
    DMIC pre-divider set in divide by 96

enumerator kDMIC_PdmDiv128
    DMIC pre-divider set in divide by 128

enum __compensation
    Pre-emphasis Filter coefficient value for 2FS and 4FS modes.

    Values:

    enumerator kDMIC_CompValueZero
        Compensation 0

    enumerator kDMIC_CompValueNegativePoint16
        Compensation -0.16

    enumerator kDMIC_CompValueNegativePoint15
        Compensation -0.15

    enumerator kDMIC_CompValueNegativePoint13
        Compensation -0.13

enum __dc_removal
    DMIC DC filter control values.

    Values:

    enumerator kDMIC_DcNoRemove
        Flat response no filter

    enumerator kDMIC_DcCut155
        Cut off Frequency is 155 Hz

    enumerator kDMIC_DcCut78
        Cut off Frequency is 78 Hz

    enumerator kDMIC_DcCut39
        Cut off Frequency is 39 Hz

enum __dmic_io
    DMIC IO configiration.

    Values:

    enumerator kDMIC_PdmDual
        Two separate pairs of PDM wires

enumerator kDMIC_PdmStereo

    Stereo data0

enumerator kDMIC_PdmBypass

    Clk Bypass clocks both channels

enumerator kDMIC_PdmBypassClk0

    Clk Bypass clocks only channel0

enumerator kDMIC_PdmBypassClk1

    Clk Bypas clocks only channel1

enum __dmic_channel

    DMIC Channel number.

    Values:

    enumerator kDMIC_Channel0

        DMIC channel 0

    enumerator kDMIC_Channel1

        DMIC channel 1

    enumerator kDMIC_ChannelMAX

        Maximum number of DMIC channels

    _dmic_channel_mask DMIC Channel mask.

    Values:

    enumerator kDMIC_EnableChannel0

        DMIC channel 0 mask

    enumerator kDMIC_EnableChannel1

        DMIC channel 1 mask

enum __dmic_phy_sample_rate

    DMIC and decimator sample rates.

    Values:

    enumerator kDMIC_PhyFullSpeed

        Decimator gets one sample per each chosen clock edge of PDM interface

    enumerator kDMIC_PhyHalfSpeed

        PDM clock to Microphone is halved, decimator receives each sample twice

typedef enum _operation_mode operation_mode_t

    DMIC different operation modes.

typedef enum _stereo_side stereo_side_t

    DMIC left/right values.

typedef enum _compensation compensation_t

    Pre-emphasis Filter coefficient value for 2FS and 4FS modes.

typedef enum _dc_removal dc_removal_t

    DMIC DC filter control values.

typedef enum _dmic_io dmic_io_t

    DMIC IO configiration.

typedef enum _*dmic_channel* dmic_channel_t
    DMIC Channel number.

typedef enum _*dmic_phy_sample_rate* dmic_phy_sample_rate_t
    DMIC and decimator sample rates.

typedef struct _*dmic_channel_config* dmic_channel_config_t
    DMIC Channel configuration structure.

typedef void (*dmic_callback_t)(void)
    DMIC Callback function.

typedef void (*dmic_hwvad_callback_t)(void)
    HWVAD Callback function.

struct _dmic_channel_config
    #include <fsl_dmic.h> DMIC Channel configuration structure.

Public Members

pdm_div_t divhfclk
    DMIC Clock pre-divider values

uint32_t osr
    oversampling rate(CIC decimation rate) for PCM

uint32_t gainshft
    4FS PCM data gain control

compensation_t preac2coef
    Pre-emphasis Filter coefficient value for 2FS

compensation_t preac4coef
    Pre-emphasis Filter coefficient value for 4FS

dc_removal_t dc_cut_level
    DMIC DC filter control values.

uint32_t post_dc_gain_reduce
    Fine gain adjustment in the form of a number of bits to downshift

dmic_phy_sample_rate_t sample_rate
    DMIC and decimator sample rates

bool saturate16bit
    Selects 16-bit saturation. 0 means results roll over if out range and do not saturate. 1 means if the result overflows, it saturates at 0xFFFF for positive overflow and 0x8000 for negative overflow.

2.9 EMC: External Memory Controller Driver

void EMC_Init(EMC_Type *base, *emc_basic_config_t* *config)
    Initializes the basic for EMC. This function ungates the EMC clock, initializes the emc system configure and enable the EMC module. This function must be called in the first step to initialize the external memory.

    Parameters
        • base – EMC peripheral base address.

- config – The EMC basic configuration.

void EMC_DynamicMemInit(EMC_Type *base, *emc_dynamic_timing_config_t* *timing, *emc_dynamic_chip_config_t* *config, uint32_t totalChips)

Initializes the dynamic memory controller. This function initializes the dynamic memory controller in external memory controller. This function must be called after EMC_Init and before accessing the external dynamic memory.

Parameters

- base – EMC peripheral base address.

- timing – The timing and latency for dynamica memory controller setting. It shall be used for all dynamica memory chips, threfore the worst timing value for all used chips must be given.

- config – The EMC dynamic memory controller chip independent configuration pointer. This configuration pointer is actually pointer to a configration array. the array number depends on the "totalChips".

- totalChips – The total dynamic memory chip numbers been used or the length of the "emc_dynamic_chip_config_t" type memory.

void EMC_StaticMemInit(EMC_Type *base, uint32_t *extWait_Ns, *emc_static_chip_config_t* *config, uint32_t totalChips)

Initializes the static memory controller. This function initializes the static memory controller in external memory controller. This function must be called after EMC_Init and before accessing the external static memory.

Parameters

- base – EMC peripheral base address.

- extWait_Ns – The extended wait timeout or the read/write transfer time. This is common for all static memory chips and set with NULL if not required.

- config – The EMC static memory controller chip independent configuration pointer. This configuration pointer is actually pointer to a configration array. the array number depends on the "totalChips".

- totalChips – The total static memory chip numbers been used or the length of the "emc_static_chip_config_t" type memory.

void EMC_Deinit(EMC_Type *base)

Deinitializes the EMC module and gates the clock. This function gates the EMC controller clock. As a result, the EMC module doesn't work after calling this function.

Parameters

- base – EMC peripheral base address.

static inline void EMC_Enable(EMC_Type *base, bool enable)

Enables/disables the EMC module.

Parameters

- base – EMC peripheral base address.

- enable – True enable EMC module, false disable.

static inline void EMC_EnableDynamicMemControl(EMC_Type *base, bool enable)

Enables/disables the EMC Dynaimc memory controller.

Parameters

- base – EMC peripheral base address.

- enable – True enable EMC dynamic memory controller, false disable.

static inline void EMC_MirrorChipAddr(EMC_Type *base, bool enable)

Enables/disables the EMC address mirror. Enable the address mirror the EMC_CS1is mirrored to both EMC_CS0 and EMC_DYCS0 memory areas. Disable the address mirror enables EMC_cS0 and EMC_DYCS0 memory to be accessed.

> **Parameters**
>
> - base – EMC peripheral base address.
>
> - enable – True enable the address mirror, false disable the address mirror.

static inline void EMC_EnterSelfRefreshCommand(EMC_Type *base, bool enable)

Enter the self-refresh mode for dynamic memory controller. This function provided self-refresh mode enter or exit for application.

> **Parameters**
>
> - base – EMC peripheral base address.
>
> - enable – True enter the self-refresh mode, false to exit self-refresh and enter the normal mode.

static inline bool EMC_IsInSelfrefreshMode(EMC_Type *base)

Get the operating mode of the EMC. This function can be used to get the operating mode of the EMC.

> **Parameters**
>
> - base – EMC peripheral base address.
>
> **Returns**
>
> The EMC in self-refresh mode if true, else in normal mode.

static inline void EMC_EnterLowPowerMode(EMC_Type *base, bool enable)

Enter/exit the low-power mode.

> **Parameters**
>
> - base – EMC peripheral base address.
>
> - enable – True Enter the low-power mode, false exit low-power mode and return to normal mode.

FSL_EMC_DRIVER_VERSION

EMC driver version.

enum _emc_static_memwidth

Define EMC memory width for static memory device.

Values:

enumerator kEMC_8BitWidth

8 bit memory width.

enumerator kEMC_16BitWidth

16 bit memory width.

enumerator kEMC_32BitWidth

32 bit memory width.

enum _emc_static_special_config

Define EMC static configuration.

Values:

enumerator kEMC_AsynchronosPageEnable
    Enable the asynchronous page mode. page length four.

enumerator kEMC_ActiveHighChipSelect
    Chip select active high.

enumerator kEMC_ByteLaneStateAllLow
    Reads/writes the respective valuie bits in BLS3:0 are low.

enumerator kEMC_ExtWaitEnable
    Extended wait enable.

enumerator kEMC_BufferEnable
    Buffer enable.

enum _emc_dynamic_device
    EMC dynamic memory device.

    Values:

    enumerator kEMC_Sdram
        Dynamic memory device: SDRAM.

    enumerator kEMC_Lpsdram
        Dynamic memory device: Low-power SDRAM.

enum _emc_dynamic_read
    EMC dynamic read strategy.

    Values:

    enumerator kEMC_NoDelay
        No delay.

    enumerator kEMC_Cmddelay
        Command delayed strategy, using EMCCLKDELAY.

    enumerator kEMC_CmdDelayPulseOneclk
        Command delayed strategy pluse one clock cycle using EMCCLKDELAY.

    enumerator kEMC_CmddelayPulsetwoclk
        Command delayed strategy pulse two clock cycle using EMCCLKDELAY.

enum _emc_endian_mode
    EMC endian mode.

    Values:

    enumerator kEMC_LittleEndian
        Little endian mode.

    enumerator kEMC_BigEndian
        Big endian mode.

enum _emc_fbclk_src
    EMC Feedback clock input source select.

    Values:

    enumerator kEMC_IntloopbackEmcclk
        Use the internal loop back from EMC_CLK output.

    enumerator kEMC_EMCFbclkInput
        Use the external EMC_FBCLK input.

typedef enum _*emc_static_memwidth* emc_static_memwidth_t
    Define EMC memory width for static memory device.

typedef enum _*emc_static_special_config* emc_static_special_config_t
    Define EMC static configuration.

typedef enum _*emc_dynamic_device* emc_dynamic_device_t
    EMC dynamic memory device.

typedef enum _*emc_dynamic_read* emc_dynamic_read_t
    EMC dynamic read strategy.

typedef enum _*emc_endian_mode* emc_endian_mode_t
    EMC endian mode.

typedef enum _*emc_fbclk_src* emc_fbclk_src_t
    EMC Feedback clock input source select.

typedef struct _*emc_dynamic_timing_config* emc_dynamic_timing_config_t
    EMC dynamic timing/delay configure structure.

typedef struct _*emc_dynamic_chip_config* emc_dynamic_chip_config_t
    EMC dynamic memory controller independent chip configuration structure. Please take
    refer to the address mapping table in the RM in EMC chapter when you set the "devAd-
    drMap". Choose the right Bit 14 Bit12 ~ Bit 7 group in the table according to the bus
    width/banks/row/colum length for you device. Set devAddrMap with the value make up
    with the seven bits (bit14 bit12 ~ bit 7) and inset the bit 13 with 0. for example, if the bit 14
    and bit12 ~ bit7 is 1000001 is choosen according to the 32bit high-performance bus width
    with 2 banks, 11 row lwngth, 8 column length. Set devAddrMap with 0x81.

typedef struct _*emc_static_chip_config* emc_static_chip_config_t
    EMC static memory controller independent chip configuration structure.

typedef struct _*emc_basic_config* emc_basic_config_t
    EMC module basic configuration structure.

    Defines the static memory controller configure structure and uses the EMC_Init() function
    to make necessary initializations.

EMC_STATIC_MEMDEV_NUM
    Define the chip numbers for dynamic and static memory devices.

EMC_DYNAMIC_MEMDEV_NUM

EMC_ADDRMAP_SHIFT

EMC_ADDRMAP_MASK

EMC_ADDRMAP(x)

EMC_HZ_ONEMHZ

EMC_MILLISECS_ONESEC

EMC_SDRAM_MODE_CL_SHIFT

EMC_SDRAM_MODE_CL_MASK

EMC_SDRAM_NOP_DELAY_US
    EDMA_SDRAM NOP command wait us.

EMC_SDRAM_PRECHARGE_DELAY_US
    EDMA_SDRAM precharge command wait us.

EMC_SDRAM_AUTO_REFRESH_DELAY_US
    EDMA_SDRAM auto refresh wait us.

struct _emc_dynamic_timing_config
    #include <fsl_emc.h> EMC dynamic timing/delay configure structure.

Public Members

uint32_t refreshPeriod_Nanosec
    The refresh period in unit of nanosecond.

uint32_t tRp_Ns
    Precharge command period in unit of nanosecond.

uint32_t tRas_Ns
    Active to precharge command period in unit of nanosecond.

uint32_t tSrex_Ns
    Self-refresh exit time in unit of nanosecond.

uint32_t tApr_Ns
    Last data out to active command time in unit of nanosecond.

uint32_t tDal_Ns
    Data-in to active command in unit of nanosecond.

uint32_t tWr_Ns
    Write recovery time in unit of nanosecond.

uint32_t tRc_Ns
    Active to active command period in unit of nanosecond.

uint32_t tRfc_Ns
    Auto-refresh period and auto-refresh to active command period in unit of nanosecond.

uint32_t tXsr_Ns
    Exit self-refresh to active command time in unit of nanosecond.

uint32_t tRrd_Ns
    Active bank A to active bank B latency in unit of nanosecond.

uint8_t tMrd_Nclk
    Load mode register to active command time in unit of EMCCLK cycles.

struct _emc_dynamic_chip_config
    #include <fsl_emc.h> EMC dynamic memory controller independent chip configuration structure. Please take refer to the address mapping table in the RM in EMC chapter when you set the "devAddrMap". Choose the right Bit 14 Bit12 ~ Bit 7 group in the table according to the bus width/banks/row/colum length for you device. Set devAddrMap with the value make up with the seven bits (bit14 bit12 ~ bit 7) and inset the bit 13 with 0. for example, if the bit 14 and bit12 ~ bit7 is 1000001 is choosen according to the 32bit high-performance bus width with 2 banks, 11 row lwngth, 8 column length. Set devAddrMap with 0x81.

Public Members

uint8_t chipIndex
    Chip Index, range from 0 ~ EMC_DYNAMIC_MEMDEV_NUM - 1.

emc_dynamic_device_t dynamicDevice
    All chips shall use the same device setting. mixed use are not supported.

uint8_t rAS_Nclk

    Active to read/write delay tRCD.

uint16_t sdramModeReg

    Sdram mode register setting.

uint16_t sdramExtModeReg

    Used for low-power sdram device. The extended mode register.

uint8_t devAddrMap

    dynamic device address mapping, choose the address mapping for your specific device.

struct _emc_static_chip_config

    #include <fsl_emc.h> EMC static memory controller independent chip configuration structure.

Public Members

emc_static_memwidth_t memWidth

    Memory width.

uint32_t specailConfig

    Static configuration,a logical OR of "emc_static_special_config_t".

uint32_t tWaitWriteEn_Ns

    The delay form chip select to write enable in unit of nanosecond.

uint32_t tWaitOutEn_Ns

    The delay from chip selcet to output enable in unit of nanosecond.

uint32_t tWaitReadNoPage_Ns

    In No-page mode, the delay from chip select to read access in unit of nanosecond.

uint32_t tWaitReadPage_Ns

    In page mode, the read after the first read wait states in unit of nanosecond.

uint32_t tWaitWrite_Ns

    The delay from chip select to write access in unit of nanosecond.

uint32_t tWaitTurn_Ns

    The Bus turn-around time in unit of nanosecond.

struct _emc_basic_config

    #include <fsl_emc.h> EMC module basic configuration structure.

    Defines the static memory controller configure structure and uses the EMC_Init() function to make necessary initializations.

Public Members

emc_endian_mode_t endian

    Endian mode .

emc_fbclk_src_t fbClkSrc

    The feedback clock source.

uint8_t emcClkDiv

    EMC_CLK = AHB_CLK / (emc_clkDiv + 1).

2.10   FLEXCOMM: FLEXCOMM Driver

2.11   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.

2.12   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

2.13 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.

2.13. GINT: Group GPIO Input Interrupt Driver

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. –

2.14 I2C: Inter-Integrated Circuit Driver

2.15 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 – Sucessully 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.
 > >
 > > • kStataus_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 lenght 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.

2.16 I2C Driver

FSL_I2C_DRIVER_VERSION
>   I2C driver version.

>   I2C status return codes.

>   *Values:*

enumerator kStatus_I2C_Busy
>   The master is already performing a transfer.

enumerator kStatus_I2C_Idle
>   The slave driver is idle.

enumerator kStatus_I2C_Nak
>   The slave device sent a NAK in response to a byte.

enumerator kStatus_I2C_InvalidParameter
>   Unable to proceed due to invalid parameter.

enumerator kStatus_I2C_BitError
>   Transferred bit was not seen on the bus.

enumerator kStatus_I2C_ArbitrationLost
>   Arbitration lost error.

enumerator kStatus_I2C_NoTransferInProgress
>   Attempt to abort a transfer when one is not in progress.

enumerator kStatus_I2C_DmaRequestFail
    DMA request failed.

enumerator kStatus_I2C_StartStopError
    Start and stop error.

enumerator kStatus_I2C_UnexpectedState
    Unexpected state.

enumerator kStatus_I2C_Timeout
    Timeout when waiting for I2C master/slave pending status to set to continue transfer.

enumerator kStatus_I2C_Addr_Nak
    NAK received for Address

enumerator kStatus_I2C_EventTimeout
    Timeout waiting for bus event.

enumerator kStatus_I2C_SclLowTimeout
    Timeout SCL signal remains low.

enum _i2c_status_flags
    I2C status flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingFlag
    The I2C module is waiting for software interaction. bit 0

enumerator kI2C_MasterArbitrationLostFlag
    The arbitration of the bus was lost. There was collision on the bus. bit 4

enumerator kI2C_MasterStartStopErrorFlag
    There was an error during start or stop phase of the transaction. bit 6

enumerator kI2C_MasterIdleFlag
    The I2C master idle status. bit 5

enumerator kI2C_MasterRxReadyFlag
    The I2C master rx ready status. bit 1

enumerator kI2C_MasterTxReadyFlag
    The I2C master tx ready status. bit 2

enumerator kI2C_MasterAddrNackFlag
    The I2C master address nack status. bit 7

enumerator kI2C_MasterDataNackFlag
    The I2C master data nack status. bit 3

enumerator kI2C_SlavePendingFlag
    The I2C module is waiting for software interaction. bit 8

enumerator kI2C_SlaveNotStretching
    Indicates whether the slave is currently stretching clock (0 = yes, 1 = no). bit 11

enumerator kI2C_SlaveSelected
    Indicates whether the slave is selected by an address match. bit 14

enumerator kI2C_SaveDeselected
    Indicates that slave was previously deselected (deselect event took place, w1c). bit 15

enumerator kI2C_SlaveAddressedFlag
    One of the I2C slave's 4 addresses is matched. bit 22

enumerator kI2C_SlaveReceiveFlag
    Slave receive data available. bit 9

enumerator kI2C_SlaveTransmitFlag
    Slave data can be transmitted. bit 10

enumerator kI2C_SlaveAddress0MatchFlag
    Slave address0 match. bit 20

enumerator kI2C_SlaveAddress1MatchFlag
    Slave address1 match. bit 12

enumerator kI2C_SlaveAddress2MatchFlag
    Slave address2 match. bit 13

enumerator kI2C_SlaveAddress3MatchFlag
    Slave address3 match. bit 21

enumerator kI2C_MonitorReadyFlag
    The I2C monitor ready interrupt. bit 16

enumerator kI2C_MonitorOverflowFlag
    The monitor data overrun interrupt. bit 17

enumerator kI2C_MonitorActiveFlag
    The monitor is active. bit 18

enumerator kI2C_MonitorIdleFlag
    The monitor idle interrupt. bit 19

enumerator kI2C_EventTimeoutFlag
    The bus event timeout interrupt. bit 24

enumerator kI2C_SclTimeoutFlag
    The SCL timeout interrupt. bit 25

enumerator kI2C_MasterAllClearFlags

enumerator kI2C_SlaveAllClearFlags

enumerator kI2C_CommonAllClearFlags

enum __i2c_interrupt_enable
    I2C interrupt enable.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingInterruptEnable
    The I2C master communication pending interrupt.

enumerator kI2C_MasterArbitrationLostInterruptEnable
    The I2C master arbitration lost interrupt.

enumerator kI2C_MasterStartStopErrorInterruptEnable
    The I2C master start/stop timing error interrupt.

enumerator kI2C_SlavePendingInterruptEnable
    The I2C slave communication pending interrupt.

enumerator kI2C_SlaveNotStretchingInterruptEnable
    The I2C slave not streching 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

2.17 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 superceded 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 indicete 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.

- kStataus_I2C_Nak – Transfer error, receive NAK during transfer.

- kStataus_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.

2.18 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 flase to disable.

static inline void I2C_SlaveClearStatusFlags(I2C_Type *base, uint32_t statusMask)

Clears the I2C status flag state.

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

See also:

_i2c_slave_flags.

Parameters

- base – The I2C peripheral base address.

- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _i2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to I2C_SlaveGetStatusFlags().

status_t I2C_SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- base – The I2C peripheral base address.

- txBuff – The pointer to the data to be transferred.

- txSize – The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been sent.

Returns

kStatus_Fail Unexpected slave state (master data write while master read from slave is expected).

status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- base – The I2C peripheral base address.

- rxBuff – The pointer to the data to be transferred.

- rxSize – The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been received.

Returns

kStatus_Fail Unexpected slave state (master data read while master write to slave is expected).

void I2C_SlaveTransferCreateHandle(I2C_Type *base, *i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)

Creates a new handle for the I2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I2C_SlaveTransferAbort() API shall be called.

Parameters

- base – The I2C peripheral base address.

- handle – **[out]** Pointer to the I2C slave driver handle.

- callback – User provided pointer to the asynchronous callback function.

- userData – User provided pointer to the application callback data.

status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, *i2c_slave_handle_t *handle, uint32_t eventMask)

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes kI2C_SlaveTransmitEvent callback. If no slave Rx transfer is busy, a master write to slave request invokes kI2C_SlaveReceiveEvent callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The I2C peripheral base address.

- handle – Pointer to i2c_slave_handle_t structure which stores the transfer state.

- eventMask – Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.

- kStatus_I2C_Busy – Slave transfers have already been started on this handle.

status_t I2C_SlaveSetSendBuffer(I2C_Type *base, volatile *i2c_slave_transfer_t* *transfer, const void *txData, size_t txSize, uint32_t eventMask)

Starts accepting master read from slave requests.

The function can be called in response to kI2C_SlaveTransmitEvent callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The I2C peripheral base address.

- transfer – Pointer to i2c_slave_transfer_t structure.

- txData – Pointer to data to send to master.

- txSize – Size of txData in bytes.

- eventMask – Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.

- kStatus_I2C_Busy – Slave transfers have already been started on this handle.

status_t I2C_SlaveSetReceiveBuffer(I2C_Type *base, volatile *i2c_slave_transfer_t* *transfer, void *rxData, size_t rxSize, uint32_t eventMask)

Starts accepting master write to slave requests.

The function can be called in response to kI2C_SlaveReceiveEvent callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The I2C peripheral base address.

- transfer – Pointer to i2c_slave_transfer_t structure.

- rxData – Pointer to data to store data from master.

- rxSize – Size of rxData in bytes.

- eventMask – Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.

- kStatus_I2C_Busy – Slave transfers have already been started on this handle.

static inline uint32_t I2C_SlaveGetReceivedAddress(I2C_Type *base, volatile *i2c_slave_transfer_t *transfer*)

Returns the slave address sent by the I2C master.

This function should only be called from the address match event callback kI2C_SlaveAddressMatchEvent.

Parameters

- base – The I2C peripheral base address.

- transfer – The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

void I2C_SlaveTransferAbort(I2C_Type *base, *i2c_slave_handle_t* *handle)

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- base – The I2C peripheral base address.

- handle – Pointer to i2c_slave_handle_t structure which stores the transfer state.

Return values

- kStatus_Success –

- kStatus_I2C_Idle –

status_t I2C_SlaveTransferGetCount(I2C_Type *base, *i2c_slave_handle_t* *handle, size_t *count)

Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.

Parameters

- base – I2C base pointer.

- handle – pointer to i2c_slave_handle_t structure.

- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.

- kStatus_Success – Successfully return the count.

void I2C_SlaveTransferHandleIRQ(I2C_Type *base, *i2c_slave_handle_t* *handle)

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I2C peripheral base address.

- handle – Pointer to i2c_slave_handle_t structure which stores the transfer state.

enum __i2c_slave_address_register

I2C slave address register.

Values:

enumerator kI2C_SlaveAddressRegister0

Slave Address 0 register.

enumerator kI2C_SlaveAddressRegister1

Slave Address 1 register.

enumerator kI2C_SlaveAddressRegister2

Slave Address 2 register.

enumerator kI2C_SlaveAddressRegister3

Slave Address 3 register.

enum __i2c_slave_address_qual_mode

I2C slave address match options.

Values:

enumerator kI2C_QualModeMask

The SLVQUAL0 field (qualAddress) is used as a logical mask for matching address0.

enumerator kI2C_QualModeExtend

The SLVQUAL0 (qualAddress) field is used to extend address 0 matching in a range of addresses.

enum __i2c_slave_bus_speed

I2C slave bus speed options.

Values:

enumerator kI2C_SlaveStandardMode

enumerator kI2C_SlaveFastMode

enumerator kI2C_SlaveFastModePlus

enumerator kI2C_SlaveHsMode

enum __i2c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kI2C_SlaveAddressMatchEvent
> Received the slave address after a start or repeated start.

enumerator kI2C_SlaveTransmitEvent
> Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kI2C_SlaveReceiveEvent
> Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI2C_SlaveCompletionEvent
> All data in the active transfer have been consumed.

enumerator kI2C_SlaveDeselectedEvent
> The slave function has become deselected (SLVSEL flag changing from 1 to 0.

enumerator kI2C_SlaveAllEvents
> Bit mask of all available events.

enum _i2c_slave_fsm
> I2C slave software finite state machine states.

> *Values:*

> enumerator kI2C_SlaveFsmAddressMatch

> enumerator kI2C_SlaveFsmReceive

> enumerator kI2C_SlaveFsmTransmit

typedef enum *_i2c_slave_address_register* i2c_slave_address_register_t
> I2C slave address register.

typedef struct *_i2c_slave_address* i2c_slave_address_t
> Data structure with 7-bit Slave address and Slave address disable.

typedef enum *_i2c_slave_address_qual_mode* i2c_slave_address_qual_mode_t
> I2C slave address match options.

typedef enum *_i2c_slave_bus_speed* i2c_slave_bus_speed_t
> I2C slave bus speed options.

typedef struct *_i2c_slave_config* i2c_slave_config_t
> Structure with settings to initialize the I2C slave module.

> This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

> The configuration structure can be made constant so it resides in flash.

typedef enum *_i2c_slave_transfer_event* i2c_slave_transfer_event_t
> Set of events sent to the callback for non blocking slave transfers.

> These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

---

typedef struct _*i2c_slave_handle* i2c_slave_handle_t
>   I2C slave handle typedef.

typedef struct _*i2c_slave_transfer* i2c_slave_transfer_t
>   I2C slave transfer structure.

typedef void (*i2c_slave_transfer_callback_t)(I2C_Type *base, volatile *i2c_slave_transfer_t* *transfer, void *userData)
>   Slave event callback function pointer type.

>   This callback is used only for the slave non-blocking transfer API. To install a callback, use the I2C_SlaveSetCallback() function after you have created a handle.

>   **Param base**
>>   Base address for the I2C instance on which the event occurred.

>   **Param transfer**
>>   Pointer to transfer descriptor containing values passed to and/or from the callback.

>   **Param userData**
>>   Arbitrary pointer-sized value passed from the application.

typedef enum _*i2c_slave_fsm* i2c_slave_fsm_t
>   I2C slave software finite state machine states.

typedef void (*flexcomm_i2c_master_irq_handler_t)(I2C_Type *base, *i2c_master_handle_t* *handle)
>   Typedef for master interrupt handler.

typedef void (*flexcomm_i2c_slave_irq_handler_t)(I2C_Type *base, *i2c_slave_handle_t* *handle)
>   Typedef for slave interrupt handler.

struct _i2c_slave_address
>   *#include <fsl_i2c.h>* Data structure with 7-bit Slave address and Slave address disable.

Public Members

uint8_t address
>   7-bit Slave address SLVADR.

bool addressDisable
>   Slave address disable SADISABLE.

struct _i2c_slave_config
>   *#include <fsl_i2c.h>* Structure with settings to initialize the I2C slave module.

>   This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

>   The configuration structure can be made constant so it resides in flash.

Public Members

i2c_slave_address_t address0
>   Slave's 7-bit address and disable.

2.18. I2C Slave Driver

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.

2.19 I2S: I2S Driver

2.20 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 callbed 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

2.21 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;
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 – seondary 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 – seondary 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 intialized 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

2.22 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.

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

2.23 INPUTMUX: Input Multiplexing Driver

enum _inputmux_connection_t
    INPUTMUX connections type.

    Values:

    enumerator kINPUTMUX_SctGpi0ToSct0
        SCT INMUX.

    enumerator kINPUTMUX_SctGpi1ToSct0

    enumerator kINPUTMUX_SctGpi2ToSct0

    enumerator kINPUTMUX_SctGpi3ToSct0

    enumerator kINPUTMUX_SctGpi4ToSct0

    enumerator kINPUTMUX_SctGpi5ToSct0

    enumerator kINPUTMUX_SctGpi6ToSct0

    enumerator kINPUTMUX_SctGpi7ToSct0

enumerator kINPUTMUX__T0Out0ToSct0

enumerator kINPUTMUX__T1Out0ToSct0

enumerator kINPUTMUX__T2Out0ToSct0

enumerator kINPUTMUX__T3Out0ToSct0

enumerator kINPUTMUX__T4Out0ToSct0

enumerator kINPUTMUX__AdcThcmpIrqToSct0

enumerator kINPUTMUX__GpioIntBmatchToSct0

enumerator kINPUTMUX__Usb0FrameToggleToSct0

enumerator kINPUTMUX__Usb1FrameToggleToSct0

enumerator kINPUTMUX__ArmTxevToSct0

enumerator kINPUTMUX__DebugHaltedToSct0

enumerator kINPUTMUX__SmartCard0TxActivreToSct0

enumerator kINPUTMUX__SmartCard0RxActivreToSct0

enumerator kINPUTMUX__SmartCard1TxActivreToSct0

enumerator kINPUTMUX__SmartCard1RxActivreToSct0

enumerator kINPUTMUX__I2s6SclkToSct0

enumerator kINPUTMUX__I2sS7clkToSct0
    Frequency measure.

enumerator kINPUTMUX__MainOscToFreqmeas

enumerator kINPUTMUX__Fro12MhzToFreqmeas

enumerator kINPUTMUX__Fro96MhzToFreqmeas

enumerator kINPUTMUX__WdtOscToFreqmeas

enumerator kINPUTMUX__32KhzOscToFreqmeas

enumerator kINPUTMUX__MainClkToFreqmeas

enumerator kINPUTMUX__FreqmeGpioClk_a

enumerator kINPUTMUX__FreqmeGpioClk_b
    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_PinInt0ToDma

enumerator kINPUTMUX_PinInt1ToDma

enumerator kINPUTMUX_PinInt2ToDma

enumerator kINPUTMUX_PinInt3ToDma

enumerator kINPUTMUX__Ctimer0M0ToDma

enumerator kINPUTMUX__Ctimer0M1ToDma

enumerator kINPUTMUX__Ctimer1M0ToDma

enumerator kINPUTMUX__Ctimer1M1ToDma

enumerator kINPUTMUX__Ctimer2M0ToDma

enumerator kINPUTMUX__Ctimer2M1ToDma

enumerator kINPUTMUX__Ctimer3M0ToDma

enumerator kINPUTMUX__Ctimer3M1ToDma

enumerator kINPUTMUX__Ctimer4M0ToDma

enumerator kINPUTMUX__Ctimer4M1ToDma

enumerator kINPUTMUX__Otrig0ToDma

enumerator kINPUTMUX__Otrig1ToDma

enumerator kINPUTMUX__Otrig2ToDma

enumerator kINPUTMUX__Otrig3ToDma

enumerator AES256_INPUT_DMA_REQ

enumerator AES256_OUTPUT_DMA_REQ
   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_DmaDmic0Ch0TrigoutToTriginChannels

enumerator kINPUTMUX_Dmamic0Ch1TrigoutToTriginChannels

enumerator kINPUTMUX_DmaSpifi0TrigoutToTriginChannels

enumerator kINPUTMUX_DmaSha_TrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm8RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm8TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm9RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm9TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaSmartcard0RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaSmartcard0TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaSmartcard1RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaSmartcard1TxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm10RxTrigoutToTriginChannels

enumerator kINPUTMUX_DmaFlexcomm10TxTrigoutToTriginChannels

typedef enum _*inputmux_connection_t* inputmux_connection_t
    INPUTMUX connections type.

SCT0_PMUX_ID
    Periphinmux IDs.

PINTSEL_PMUX_ID

DMA_TRIG0_PMUX_ID

DMA_OTRIG_PMUX_ID

FREQMEAS_PMUX_ID

PMUX_SHIFT

FSL_INPUTMUX_DRIVER_VERSION
    Group interrupt driver version for SDK.

void INPUTMUX_Init(void *base)
    Initialize INPUTMUX peripheral.

    This function enables the INPUTMUX clock.

    Parameters
        • base – Base address of the INPUTMUX peripheral.

    Return values
        None. –

void INPUTMUX_AttachSignal(void *base, uint16_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 INPUT-MUX 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. –

2.24 Common Driver

FSL_COMMON_DRIVER_VERSION
> common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE
> No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART
> Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART
> Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
> Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
> Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM
> Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART
> Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART
> Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART
> Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(a, b)

Computes the minimum of *a* and *b*.

MAX(a, b)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true** , else return **false** .

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

SDK_SIZEALIGN(var, alignbytes)

    Macro to define a variable with L1 d-cache line size alignment

    Macro to define a variable with L2 cache line size alignment

    Macro to change a value to a given size aligned value (rounded up)

SDK_SIZEALIGN_UP(var, alignbytes)

    Macro to change a value to a given size aligned value (rounded up), the wrapper of SDK_SIZEALIGN

SDK_SIZEALIGN_DOWN(var, alignbytes)

    Macro to change a value to a given size aligned value (rounded down)

SDK_IS_ALIGNED(var, alignbytes)

    Macro to check if a value is aligned to a given size

AT_NONCACHEABLE_SECTION(var)

    Define a variable *var*, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

    Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_NONCACHEABLE_SECTION_INIT(var)

    Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

    Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_CACHE_LINE_SECTION(var)

    Define a variable *var,* which is cache line size aligned and be placed in CacheLineData section.

AT_CACHE_LINE_SECTION_INIT(var)

    Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

AT_QUICKACCESS_SECTION_CODE(func)

    Place function in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA(var)

    Place data in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA_ALIGN(var, alignbytes)

    Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

MCUX_RAMFUNC

    Function attribute to place function in RAM. For example, to place function my_func in ram, use like:

```
MCUX_RAMFUNC my_func
```

RAMFUNCTION_SECTION_CODE(func)

    Place function in ram.

enum __status_groups

    Status group numbers.

    Values:

enumerator kStatusGroup_Generic
    Group number for generic status codes.

enumerator kStatusGroup_FLASH
    Group number for FLASH status codes.

enumerator kStatusGroup_LPSPI
    Group number for LPSPI status codes.

enumerator kStatusGroup_FLEXIO_SPI
    Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI
    Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
    Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
    Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
    Group number for LPI2C status codes.

enumerator kStatusGroup_UART
    Group number for UART status codes.

enumerator kStatusGroup_I2C
    Group number for UART status codes.

enumerator kStatusGroup_LPSCI
    Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
    Group number for LPUART status codes.

enumerator kStatusGroup_SPI
    Group number for SPI status code.

enumerator kStatusGroup_XRDC
    Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
    Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
    Group number for SDHC status code

enumerator kStatusGroup_SDMMC
    Group number for SDMMC status code

enumerator kStatusGroup_SAI
    Group number for SAI status code

enumerator kStatusGroup_MCG
    Group number for MCG status codes.

enumerator kStatusGroup_SCG
    Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
    Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
    Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
    Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
    Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
    Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
    Group number for I2S status codes

enumerator kStatusGroup_IUART
    Group number for IUART status codes

enumerator kStatusGroup_CSI
    Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
    Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
    Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
    Group number for POWER status codes.

enumerator kStatusGroup_ENET
    Group number for ENET status codes.

enumerator kStatusGroup_PHY
    Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
    Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
    Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
    Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
    Group number for QSPI status codes.

enumerator kStatusGroup_DMA
    Group number for DMA status codes.

enumerator kStatusGroup_EDMA
    Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
    Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
    Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
    Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
    Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
    Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
    Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
    Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
    Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
    Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
    Group number for OTP status codes.

enumerator kStatusGroup_MCAN
    Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
    Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
    Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
    Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
    Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
    Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
    Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
    Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
    Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
    Group number for MMDC status codes.

enumerator kStatusGroup_PDM
    Group number for MIC status codes.

enumerator kStatusGroup_SDMA
    Group number for SDMA status codes.

enumerator kStatusGroup_ICS
    Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
    Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
    Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
    Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
    Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
    Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
    Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
    Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
    Group number for debug console status codes.

enumerator kStatusGroup_SEMC
    Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
    Starting number for application groups.

enumerator kStatusGroup_IAP
    Group number for IAP status codes

enumerator kStatusGroup_SFA
    Group number for SFA status codes

enumerator kStatusGroup_SPC
    Group number for SPC status codes.

enumerator kStatusGroup_PUF
    Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
    Group number for touch panel status codes

enumerator kStatusGroup_VBAT
    Group number for VBAT status codes

enumerator kStatusGroup_XSPI
    Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
    Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
    Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
    Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
    Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
    Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
   Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
   Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
   Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
   Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
   Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
   Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
   Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
   Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
   Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
   Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
   Group number for LED status codes.

enumerator kStatusGroup_BUTTON
   Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
   Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
   Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
   Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
   Group number for List status codes.

enumerator kStatusGroup_OSA
   Group number for OSA status codes.

enumerator kStatusGroup_COMMON_TASK
   Group number for Common task status codes.

enumerator kStatusGroup_MSG
   Group number for messaging status codes.

enumerator kStatusGroup_SDK_OCOTP
   Group number for OCOTP status codes.

enumerator kStatusGroup_SDK_FLEXSPINOR
   Group number for FLEXSPINOR status codes.

enumerator kStatusGroup_CODEC
    Group number for codec status codes.

enumerator kStatusGroup_ASRC
    Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
    Group number for codec status codes.

enumerator kStatusGroup_SDIOSLV
    Group number for SDIOSLV status codes.

enumerator kStatusGroup_MECC
    Group number for MECC status codes.

enumerator kStatusGroup_ENET_QOS
    Group number for ENET_QOS status codes.

enumerator kStatusGroup_LOG
    Group number for LOG status codes.

enumerator kStatusGroup_I3CBUS
    Group number for I3CBUS status codes.

enumerator kStatusGroup_QSCI
    Group number for QSCI status codes.

enumerator kStatusGroup_ELEMU
    Group number for ELEMU status codes.

enumerator kStatusGroup_QUEUEDSPI
    Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
    Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
    Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
    Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
    Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
    Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
    Group number for CLIF status codes.

enumerator kStatusGroup_BMA
    Group number for BMA status codes.

enumerator kStatusGroup_NETC
    Group number for NETC status codes.

enumerator kStatusGroup_ELE
    Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
    Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
    Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
    Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
    Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
    Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
    Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
    Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
    Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
    Group number for A-format status codes.

enumerator kStatusGroup_LPC_QSPI
    Group number for LPC QSPI 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 internaly).

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 internaly).

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 wil be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, "SUPPRESS_FALL_THROUGH_WARNING();" need to be added at the end of each case section which misses "break;"statement.

MSDK_REG_SECURE_ADDR(x)

    Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

    Convert the register address to the one used in non-secure mode.

MSDK_HAS_DWT_CYCCNT

    The chip supports DWT CYCCNT or not.

MSDK_INVALID_IRQ_HANDLER

    Invalid IRQ handler address.

2.25 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 then 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 cntinuously 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 cnversions 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 bass 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 cnversions 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 bass address.

bool ADC_GetConvSeqAGlobalConversionResult(ADC_Type *base, *adc_result_info_t* *info)

> Get the global ADC conversion infomation 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 infomation 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 threshhold 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 threshhold 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 channal, 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 glabal 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 comparision 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

Definfe 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 <= conversion value <= 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_inforesultshift

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 Temerature 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
Extand ADC sampling time according to source impedance 1: 0.621 kOhm.

enumerator kADC_Impedance55kOhm
Extand ADC sampling time according to source impedance 20 (default): 55 kOhm.

enumerator kADC_Impedance87kOhm
Extand 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
Definfe 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_inforesultshift* adc_inforesult_t

> Define the info result mode of different resolution.

typedef enum _*adc_tempsensor_common_mode* adc_tempsensor_common_mode_t

> Define common modes for Temerature 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 interrpt/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.

2.26   ENET: Ethernet Driver

void ENET_GetDefaultConfig(*enet_config_t* *config)

> Gets the ENET default configuration structure.

> The purpose of this API is to get the default ENET configure structure for ENET_Init(). User may use the initialized structure unchanged in ENET_Init(), or modify some fields of the structure before calling ENET_Init(). Example:

```
enet_config_t config;
ENET_GetDefaultConfig(&config);
```

> ### Parameters

> > • config – The ENET mac controller configuration structure pointer.

void ENET_Init(ENET_Type *base, const *enet_config_t* *config, uint8_t *macAddr, uint32_t refclkSrc_Hz)

> Initializes the ENET module.

> This function ungates the module clock and initializes it with the ENET basic configuration.

Note: As our transactional transmit API use the zero-copy transmit buffer. So there are two thing we emphasize here:

> a. Tx buffer free/requeue for application should be done in the Tx interrupt handler. Please set callback: kENET_TxIntEvent with Tx buffer free/requeue process APIs.

> b. The Tx interrupt is forced to open.

> ### Parameters

> > • base – ENET peripheral base address.

> > • config – ENET mac configuration structure pointer. The "enet_config_t" type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.

> > • macAddr – ENET mac address of Ethernet device. This MAC address should be provided.

> > • refclkSrc_Hz – ENET input reference clock.

void ENET_Deinit(ENET_Type *base)

> Deinitializes the ENET module.

> This function gates the module clock and disables the ENET module.

> ### Parameters

> > • base – ENET peripheral base address.

status_t ENET_DescriptorInit(ENET_Type *base, *enet_config_t* *config, *enet_buffer_config_t* *bufferConfig)

> Initialize for all ENET descriptors.

Note: This function finishes all Tx/Rx descriptors initialization. The descriptor initialization should be called after ENET_Init().

> ### Parameters

- base – ENET peripheral base address.

- config – The configuration for ENET.

- bufferConfig – All buffers configuration.

status_t ENET_RxBufferAllocAll(ENET_Type *base, *enet_handle_t* *handle)

Allocates Rx buffers for all BDs. It's used for zero copy Rx. In zero copy Rx case, Rx buffers are dynamic. This function will populate initial buffers in all BDs for receiving. Then ENET_GetRxFrame() is used to get Rx frame with zero copy, it will allocate new buffer to replace the buffer in BD taken by application, application should free those buffers after they're used.

Note: This function should be called after ENET_CreateHandler() and buffer allocating callback function should be ready.

Parameters

- base – ENET peripheral base address.

- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.

void ENET_RxBufferFreeAll(ENET_Type *base, *enet_handle_t* *handle)

Frees Rx buffers in all BDs. It's used for zero copy Rx. In zero copy Rx case, Rx buffers are dynamic. This function will free left buffers in all BDs.

Parameters

- base – ENET peripheral base address.

- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.

void ENET_StartRxTx(ENET_Type *base, uint8_t txRingNum, uint8_t rxRingNum)

Starts the ENET Tx/Rx. This function enable the Tx/Rx and starts the Tx/Rx DMA. This shall be set after ENET initialization and before starting to receive the data.

Note: This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

Parameters

- base – ENET peripheral base address.

- rxRingNum – The number of the used Rx rings. It shall not be larger than the ENET_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.

- txRingNum – The number of the used Tx rings. It shall not be larger than the ENET_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.

void ENET_SetISRHandler(ENET_Type *base, *enet_isr_t* ISRHandler)

Set the second level IRQ handler.

Parameters

- base – ENET peripheral base address.

- ISRHandler – The handler to install.

static inline void ENET_SetMII(ENET_Type *base, *enet_mii_speed_t* speed, *enet_mii_duplex_t* duplex)

>   Sets the ENET MII speed and duplex.

>   This API is provided to dynamically change the speed and dulpex for MAC.

> > **Parameters**

> > > • base – ENET peripheral base address.

> > > • speed – The speed of the RMII mode.

> > > • duplex – The duplex of the RMII mode.

void ENET_SetSMI(ENET_Type *base)

>   Sets the ENET SMI(serial management interface)- MII management interface.

> > **Parameters**

> > > • base – ENET peripheral base address.

static inline bool ENET_IsSMIBusy(ENET_Type *base)

>   Checks if the SMI is busy.

> > **Parameters**

> > > • base – ENET peripheral base address.

> > **Returns**

> > > The status of MII Busy status.

static inline uint16_t ENET_ReadSMIData(ENET_Type *base)

>   Reads data from the PHY register through SMI interface.

> > **Parameters**

> > > • base – ENET peripheral base address.

> > **Returns**

> > > The data read from PHY

void ENET_StartSMIWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t data)

>   Sends the MDIO IEEE802.3 Clause 22 format write command.

> > **Parameters**

> > > • base – ENET peripheral base address.

> > > • phyAddr – The PHY address.

> > > • regAddr – The PHY register.

> > > • data – The data written to PHY.

void ENET_StartSMIRead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr)

>   Sends the MDIO IEEE802.3 Clause 22 format read command.

> > **Parameters**

> > > • base – ENET peripheral base address.

> > > • phyAddr – The PHY address.

> > > • regAddr – The PHY register.

status_t ENET_MDIOWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t data)

>   MDIO write with IEEE802.3 Clause 22 format.

> > **Parameters**

> > > • base – ENET peripheral base address.

- phyAddr – The PHY address.

- regAddr – The PHY register.

- data – The data written to PHY.

Returns
kStatus_Success MDIO access succeeds.

Returns
kStatus_Timeout MDIO access timeout.

status_t ENET_MDIORead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t *pData)

MDIO read with IEEE802.3 Clause 22 format.

Parameters

- base – ENET peripheral base address.

- phyAddr – The PHY address.

- regAddr – The PHY register.

- pData – The data read from PHY.

Returns
kStatus_Success MDIO access succeeds.

Returns
kStatus_Timeout MDIO access timeout.

uint32_t ENET_GetInstance(ENET_Type *base)

Get the ENET instance from peripheral base address.

Parameters

- base – ENET peripheral base address.

Returns
ENET instance.

static inline void ENET_SetMacAddr(ENET_Type *base, uint8_t *macAddr)

Sets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.

- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

void ENET_GetMacAddr(ENET_Type *base, uint8_t *macAddr)

Gets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.

- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

static inline void ENET_AcceptAllMulticast(ENET_Type *base)

Enable ENET device to accept all multicast frames.

Parameters

- base – ENET peripheral base address.

static inline void ENET_RejectAllMulticast(ENET_Type *base)

>   ENET device reject to accept all multicast frames.

>   **Parameters**

>   >   • base – ENET peripheral base address.

void ENET_EnterPowerDown(ENET_Type *base, uint32_t *wakeFilter)

>   Set the MAC to enter into power down mode. the remote power wake up frame and magic frame can wake up the ENET from the power down mode.

>   **Parameters**

>   >   • base – ENET peripheral base address.

>   >   • wakeFilter – The wakeFilter provided to configure the wake up frame fitlter. Set the wakeFilter to NULL is not required. But if you have the filter requirement, please make sure the wakeFilter pointer shall be eight continous 32-bits configuration.

static inline void ENET_ExitPowerDown(ENET_Type *base)

>   Set the MAC to exit power down mode. Eixt from the power down mode and recover to normal work mode.

>   **Parameters**

>   >   • base – ENET peripheral base address.

void ENET_EnableInterrupts(ENET_Type *base, uint32_t mask)

>   Enables the ENET DMA and MAC interrupts.

>   This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t. For example, to enable the dma and mac interrupt, do the following.

```
ENET_EnableInterrupts(ENET, kENET_DmaRx | kENET_DmaTx | kENET_MacPmt);
```

>   **Parameters**

>   >   • base – ENET peripheral base address.

>   >   • mask – ENET interrupts to enable. This is a logical OR of both enumeration :: enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t.

void ENET_DisableInterrupts(ENET_Type *base, uint32_t mask)

>   Disables the ENET DMA and MAC interrupts.

>   This function disables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t. For example, to disable the dma and mac interrupt, do the following.

```
ENET_DisableInterrupts(ENET, kENET_DmaRx | kENET_DmaTx | kENET_MacPmt);
```

>   **Parameters**

>   >   • base – ENET peripheral base address.

>   >   • mask – ENET interrupts to disables. This is a logical OR of both enumeration :: enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t.

static inline uint32_t ENET_GetDmaInterruptStatus(ENET_Type *base, uint8_t channel)

>   Gets the ENET DMA interrupt status flag.

>   **Parameters**

>   >   • base – ENET peripheral base address.

- channel – The DMA Channel. Shall not be larger than ENET_RING_NUM_MAX.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: enet_dma_interrupt_enable_t.

static inline void ENET_ClearDmaInterruptStatus(ENET_Type *base, uint8_t channel, uint32_t mask)

Clear the ENET DMA interrupt status flag.

Parameters

- base – ENET peripheral base address.

- channel – The DMA Channel. Shall not be larger than ENET_RING_NUM_MAX.

- mask – The event status of the interrupt source. This is the logical OR of members of the enumeration :: enet_dma_interrupt_enable_t.

static inline uint32_t ENET_GetMacInterruptStatus(ENET_Type *base)

Gets the ENET MAC interrupt status flag.

Parameters

- base – ENET peripheral base address.

Returns

The event status of the interrupt source. Use the enum in enet_mac_interrupt_enable_t and right shift ENET_MACINT_ENUM_OFFSET to mask the returned value to get the exact interrupt status.

void ENET_ClearMacInterruptStatus(ENET_Type *base, uint32_t mask)

Clears the ENET mac interrupt events status flag.

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the enet_mac_interrupt_enable_t. For example, to clear the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_ClearMacInterruptStatus(ENET, kENET_MacPmt);
```

Parameters

- base – ENET peripheral base address.

- mask – ENET interrupt source to be cleared. This is the logical OR of members of the enumeration :: enet_mac_interrupt_enable_t.

static inline bool ENET_IsTxDescriptorDmaOwn(*enet_tx_bd_struct_t* *txDesc)

Get the Tx descriptor DMA Own flag.

Parameters

- txDesc – The given Tx descriptor.

Return values

True – the dma own Tx descriptor, false application own Tx descriptor.

void ENET_SetupTxDescriptor(*enet_tx_bd_struct_t* *txDesc, void *buffer1, uint32_t bytes1, void *buffer2, uint32_t bytes2, uint32_t framelen, bool intEnable, bool tsEnable, *enet_desc_flag_t* flag, uint8_t slotNum)

Setup a given Tx descriptor. This function is a low level functional API to setup or prepare a given Tx descriptor.

Note: This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required. Transmit buffers are 'zero-copy' buffers, so the buffer must remain in memory until the packet has been fully transmitted. The buffers should be free or requeued in the transmit interrupt irq handler.

> **Parameters**
>
> - txDesc – The given Tx descriptor.
> - buffer1 – The first buffer address in the descriptor.
> - bytes1 – The bytes in the fist buffer.
> - buffer2 – The second buffer address in the descriptor.
> - bytes2 – The bytes in the second buffer.
> - framelen – The length of the frame to be transmitted.
> - intEnable – Interrupt enable flag.
> - tsEnable – The timestamp enable.
> - flag – The flag of this Tx desciriptor, see "enet_desc_flag_t" .
> - slotNum – The slot num used for AV mode only.

static inline void ENET_UpdateTxDescriptorTail(ENET_Type *base, uint8_t channel, uint32_t txDescTailAddrAlign)

Update the Tx descriptor tail pointer. This function is a low level functional API to update the the Tx descriptor tail. This is called after you setup a new Tx descriptor to update the tail pointer to make the new descritor accessable by DMA.

> **Parameters**
>
> - base – ENET peripheral base address.
> - channel – The Tx DMA channel.
> - txDescTailAddrAlign – The new Tx tail pointer address.

static inline void ENET_UpdateRxDescriptorTail(ENET_Type *base, uint8_t channel, uint32_t rxDescTailAddrAlign)

Update the Rx descriptor tail pointer. This function is a low level functional API to update the the Rx descriptor tail. This is called after you setup a new Rx descriptor to update the tail pointer to make the new descritor accessable by DMA and to anouse the Rx poll command for DMA.

> **Parameters**
>
> - base – ENET peripheral base address.
> - channel – The Rx DMA channel.
> - rxDescTailAddrAlign – The new Rx tail pointer address.

static inline uint32_t ENET_GetRxDescriptor(*enet_rx_bd_struct_t* *rxDesc)

Gets the context in the ENET Rx descriptor. This function is a low level functional API to get the the status flag from a given Rx descriptor.

Note: This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

> **Parameters**

- rxDesc – The given Rx descriptor.

Return values

The – RDES3 regions for write-back format Rx buffer descriptor.

void ENET_UpdateRxDescriptor(*enet_rx_bd_struct_t* *rxDesc, void *buffer1, void *buffer2, bool intEnable, bool doubleBuffEnable)

Updates the buffers and the own status for a given Rx descriptor. This function is a low level functional API to Updates the buffers and the own status for a given Rx descriptor.

Note: This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

Parameters

- rxDesc – The given Rx descriptor.
- buffer1 – The first buffer address in the descriptor.
- buffer2 – The second buffer address in the descriptor.
- intEnable – Interrupt enable flag.
- doubleBuffEnable – The double buffer enable flag.

void ENET_CreateHandler(ENET_Type *base, *enet_handle_t* *handle, *enet_config_t* *config, *enet_buffer_config_t* *bufferConfig, *enet_callback_t* callback, void *userData)

Create ENET Handler.

This is a transactional API and it's provided to store all datas which are needed during the whole transactional process. This API should not be used when you use functional APIs to do data Tx/Rx. This is funtion will store many data/flag for transactional use.

Parameters

- base – ENET peripheral base address.
- handle – ENET handler.
- config – ENET configuration.
- bufferConfig – ENET buffer configuration.
- callback – The callback function.
- userData – The application data.

status_t ENET_GetRxFrameSize(ENET_Type *base, *enet_handle_t* *handle, uint32_t *length, uint8_t channel)

Gets the size of the read frame. This function gets a received frame size from the ENET buffer descriptors.

Note: The FCS of the frame is automatically removed by MAC and the size is the length without the FCS. After calling ENET_GetRxFrameSize, ENET_ReadFrame() should be called to update the receive buffers If the result is not "kStatus_ENET_RxFrameEmpty".

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.

- length – The length of the valid frame received.

- channel – The DMAC channel for the Rx.

Return values

- kStatus_ENET_RxFrameEmpty – No frame received. Should not call ENET_ReadFrame to read frame.

- kStatus_ENET_RxFrameError – Data error happens. ENET_ReadFrame should be called with NULL data and NULL length to update the receive buffers.

- kStatus_Success – Receive a frame Successfully then the ENET_ReadFrame should be called with the right data buffer and the captured data length input.

status_t ENET_ReadFrame(ENET_Type *base, *enet_handle_t* *handle, uint8_t *data, uint32_t length, uint8_t channel, *enet_ptp_time_t* *timestamp)

Reads a frame from the ENET device. This function reads a frame from the ENET DMA descriptors. The ENET_GetRxFrameSize should be used to get the size of the prepared data buffer. For example use Rx dma channel 0:

```
uint32_t length;
enet_handle_t g_handle;
Comment: Get the received frame size firstly.
status = ENET_GetRxFrameSize(&g_handle, &length, 0);
if (length != 0)
{
    Comment: Allocate memory here with the size of "length"
    uint8_t *data = memory allocate interface;
    if (!data)
    {
        ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0);
    }
    else
    {
        status = ENET_ReadFrame(ENET, &g_handle, data, length, 0);
    }
}
else if (status == kStatus_ENET_RxFrameError)
{
    Comment: Update the received buffer when a error frame is received.
    ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0);
}
```

Parameters

- base – ENET peripheral base address.

- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.

- data – The data buffer provided by user to store the frame which memory size should be at least "length".

- length – The size of the data buffer which is still the length of the received frame.

- channel – The Rx DMA channel. Shall not be larger than 2.

- timestamp – The timestamp address to store received timestamp.

Returns
The execute status, successful or failure.

status_t ENET_GetRxFrame(ENET_Type *base, *enet_handle_t* *handle, *enet_rx_frame_struct_t*
*rxFrame, uint8_t channel)

Receives one frame in specified BD ring with zero copy.

This function will use the user-defined allocate and free callback. Every time application gets one frame through this function, driver will allocate new buffers for the BDs whose buffers have been taken by application.

Note: This function will drop current frame and update related BDs as available for DMA if new buffers allocating fails. Application must provide a memory pool including at least BD number + 1 buffers(+2 if enable double buffer) to make this function work normally.

Parameters

- base – ENET peripheral base address.

- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.

- rxFrame – The received frame information structure provided by user.

- channel – The Rx DMA channel. Shall not be larger than 2.

Return values

- kStatus_Success – Succeed to get one frame and allocate new memory for Rx buffer.

- kStatus_ENET_RxFrameEmpty – There's no Rx frame in the BD.

- kStatus_ENET_RxFrameError – There's issue in this receiving. In this function, issue frame will be dropped.

- kStatus_ENET_RxFrameDrop – There's no new buffer memory for BD, dropped this frame.

status_t ENET_SendFrame(ENET_Type *base, *enet_handle_t* *handle, *enet_tx_frame_struct_t*
*txFrame, uint8_t channel)

Transmits an ENET frame.

Note: The CRC is automatically appended to the data. Input the data to send without the CRC. This API uses input buffer for Tx, application should reclaim the buffer after Tx is over.

Parameters

- base – ENET peripheral base address.

- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.

- txFrame – The Tx frame structure.

- channel – Channel to send the frame, same with queue index.

Return values

- kStatus_Success – Send frame succeed.

- kStatus_ENET_TxFrameBusy – Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added

after each call return with kStatus_ENET_TxFrameBusy. Also need to pay attention to reclaim Tx frame after Tx is over.

- kStatus_ENET_TxFrameOverLen – Transmit frme length exceeds the 0x3FFF limit defined by the driver.

void ENET_ReclaimTxDescriptor(ENET_Type *base, *enet_handle_t* *handle, uint8_t channel)

Reclaim Tx descriptors. This function is used to update the Tx descriptor status and store the Tx timestamp when the 1588 feature is enabled. This is called by the transmit interupt IRQ handler after the complete of a frame transmission.

> **Parameters**
>
> - base – ENET peripheral base address.
> - handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
> - channel – The Tx DMA channnel.

void ENET_IRQHandler(ENET_Type *base, *enet_handle_t* *handle)

The ENET IRQ handler.

> **Parameters**
>
> - base – ENET peripheral base address.
> - handle – The ENET handler pointer.

FSL_ENET_DRIVER_VERSION

Defines the driver version.

ENET_RXDESCRIP_RD_BUFF1VALID_MASK

Defines for read format.

Buffer1 address valid.

ENET_RXDESCRIP_RD_BUFF2VALID_MASK

Buffer2 address valid.

ENET_RXDESCRIP_RD_IOC_MASK

Interrupt enable on complete.

ENET_RXDESCRIP_RD_OWN_MASK

Own bit.

ENET_RXDESCRIP_WR_ERR_MASK

Defines for write back format.

ENET_RXDESCRIP_WR_PYLOAD_MASK

ENET_RXDESCRIP_WR_PTPMSGTYPE_MASK

ENET_RXDESCRIP_WR_PTPTYPE_MASK

ENET_RXDESCRIP_WR_PTPVERSION_MASK

ENET_RXDESCRIP_WR_PTPTSA_MASK

ENET_RXDESCRIP_WR_PACKETLEN_MASK

ENET_RXDESCRIP_WR_ERRSUM_MASK

ENET_RXDESCRIP_WR_TYPE_MASK

ENET_RXDESCRIP_WR_DE_MASK

ENET_RXDESCRIP_WR_RE_MASK

ENET_RXDESCRIP_WR_OE_MASK

ENET_RXDESCRIP_WR_RS0V_MASK

ENET_RXDESCRIP_WR_RS1V_MASK

ENET_RXDESCRIP_WR_RS2V_MASK

ENET_RXDESCRIP_WR_LD_MASK

ENET_RXDESCRIP_WR_FD_MASK

ENET_RXDESCRIP_WR_CTXT_MASK

ENET_RXDESCRIP_WR_OWN_MASK

ENET_TXDESCRIP_RD_BL1_MASK
> Defines for read format.

ENET_TXDESCRIP_RD_BL2_MASK

ENET_TXDESCRIP_RD_BL1(n)

ENET_TXDESCRIP_RD_BL2(n)

ENET_TXDESCRIP_RD_TTSE_MASK

ENET_TXDESCRIP_RD_IOC_MASK

ENET_TXDESCRIP_RD_FL_MASK

ENET_TXDESCRIP_RD_FL(n)

ENET_TXDESCRIP_RD_CIC(n)

ENET_TXDESCRIP_RD_TSE_MASK

ENET_TXDESCRIP_RD_SLOT(n)

ENET_TXDESCRIP_RD_SAIC(n)

ENET_TXDESCRIP_RD_CPC(n)

ENET_TXDESCRIP_RD_LDFD(n)

ENET_TXDESCRIP_RD_LD_MASK

ENET_TXDESCRIP_RD_FD_MASK

ENET_TXDESCRIP_RD_CTXT_MASK

ENET_TXDESCRIP_RD_OWN_MASK

ENET_TXDESCRIP_WB_TTSS_MASK
> Defines for write back format.

ENET_ABNORM_INT_MASK

ENET_NORM_INT_MASK

ENET_FRAME_MAX_FRAMELEN
> Default maximum ethernet frame size.

ENET_FCS_LEN
    Ethernet Rx frame FCS length.

ENET_ADDR_ALIGNMENT
    Recommended ethernet buffer alignment.

ENET_BUFF_ALIGNMENT
    Receive buffer alignment shall be 4bytes-aligned.

ENET_RING_NUM_MAX
    The maximum number of Tx/Rx descriptor rings.

ENET_MTL_RXFIFOSIZE
    The Rx fifo size.

ENET_MTL_TXFIFOSIZE
    The Tx fifo size.

ENET_MACINT_ENUM_OFFSET
    The offset for mac interrupt in enum type.

ENET_FRAME_TX_LEN_LIMITATION
    The Tx frame length software limitation.

ENET_FRAME_RX_ERROR_BITS(x)
    The Rx frame error bits field.

    Defines the status return codes for transaction.

    Values:

    enumerator kStatus_ENET_InitMemoryFail
        Status code 4000. Init failed since buffer memory was not enough.

    enumerator kStatus_ENET_RxFrameError
        Status code 4001. A frame received but data error occurred.

    enumerator kStatus_ENET_RxFrameFail
        Status code 4002. Failed to receive a frame.

    enumerator kStatus_ENET_RxFrameEmpty
        Status code 4003. No frame arrived.

    enumerator kStatus_ENET_RxFrameDrop
        Status code 4004. Rx frame was dropped since there's no buffer memory.

    enumerator kStatus_ENET_TxFrameBusy
        Status code 4005. There were no resources for Tx operation.

    enumerator kStatus_ENET_TxFrameFail
        Status code 4006. Transmit frame failed.

    enumerator kStatus_ENET_TxFrameOverLen
        Status code 4007. Failed to send an oversize frame.

enum __enet_mii_mode
    Defines the MII/RMII mode for data interface between the MAC and the PHY.

    Values:

    enumerator kENET_MiiMode
        MII mode for data interface.

enumerator kENET_RmiiMode
> RMII mode for data interface.

enum __enet_mii_speed
> Defines the 10/100 Mbps speed for the MII data interface.
>
> *Values:*

enumerator kENET_MiiSpeed10M
> Speed 10 Mbps.

enumerator kENET_MiiSpeed100M
> Speed 100 Mbps.

enum __enet_mii_duplex
> Defines the half or full duplex for the MII data interface.
>
> *Values:*

enumerator kENET_MiiHalfDuplex
> Half duplex mode.

enumerator kENET_MiiFullDuplex
> Full duplex mode.

enum __enet_mii_normal_opcode
> Define the MII opcode for normal MDIO_CLAUSES_22 Frame.
>
> *Values:*

enumerator kENET_MiiWriteFrame
> Write frame operation for a valid MII management frame.

enumerator kENET_MiiReadFrame
> Read frame operation for a valid MII management frame.

enum __enet_dma_burstlen
> Define the DMA maximum transmit burst length.
>
> *Values:*

enumerator kENET_BurstLen1
> DMA burst length 1.

enumerator kENET_BurstLen2
> DMA burst length 2.

enumerator kENET_BurstLen4
> DMA burst length 4.

enumerator kENET_BurstLen8
> DMA burst length 8.

enumerator kENET_BurstLen16
> DMA burst length 16.

enumerator kENET_BurstLen32
> DMA burst length 32.

enumerator kENET_BurstLen64
> DMA burst length 64. eight times enabled.

enumerator kENET_BurstLen128
> DMA burst length 128. eight times enabled.

enumerator kENET_BurstLen256
> DMA burst length 256. eight times enabled.

enum __enet_desc_flag
> Define the flag for the descriptor.
>
> *Values:*
>
> enumerator kENET_MiddleFlag
>> It's a middle descriptor of the frame.
>
> enumerator kENET_LastFlagOnly
>> It's the last descriptor of the frame.
>
> enumerator kENET_FirstFlagOnly
>> It's the first descriptor of the frame.
>
> enumerator kENET_FirstLastFlag
>> It's the first and last descriptor of the frame.

enum __enet_systime_op
> Define the system time adjust operation control.
>
> *Values:*
>
> enumerator kENET_SystimeAdd
>> System time add to.
>
> enumerator kENET_SystimeSubtract
>> System time subtract.

enum __enet_ts_rollover_type
> Define the system time rollover control.
>
> *Values:*
>
> enumerator kENET_BinaryRollover
>> System time binary rollover.
>
> enumerator kENET_DigitalRollover
>> System time digital rollover.

enum __enet_special_config
> Defines some special configuration for ENET.
>
> These control flags are provided for special user requirements. Normally, these is no need to set this control flags for ENET initialization. But if you have some special requirements, set the flags to specialControl in the enet_config_t.

Note: "kENET_StoreAndForward" is recommended to be set when the ENET_PTP1588FEATURE_REQUIRED is defined or else the timestamp will be mess-up when the overflow happens.

> *Values:*
>
> enumerator kENET_DescDoubleBuffer
>> The double buffer is used in the Tx/Rx descriptor.
>
> enumerator kENET_StoreAndForward
>> The Rx/Tx store and forward enable.

---

enumerator kENET_PromiscuousEnable
    The promiscuous enabled.

enumerator kENET_FlowControlEnable
    The flow control enabled.

enumerator kENET_BroadCastRxDisable
    The broadcast disabled.

enumerator kENET_MulticastAllEnable
    All multicast are passed.

enumerator kENET_8023AS2KPacket
    8023as support for 2K packets.

enumerator kENET_RxChecksumOffloadEnable
    The Rx checksum offload enabled.

enum _enet_dma_interrupt_enable
    List of DMA interrupts supported by the ENET interrupt. This enumeration uses one-hot encoding to allow a logical OR of multiple members.

    Values:

    enumerator kENET_DmaTx
        Tx interrupt.

    enumerator kENET_DmaTxStop
        Tx stop interrupt.

    enumerator kENET_DmaTxBuffUnavail
        Tx buffer unavailable.

    enumerator kENET_DmaRx
        Rx interrupt.

    enumerator kENET_DmaRxBuffUnavail
        Rx buffer unavailable.

    enumerator kENET_DmaRxStop
        Rx stop.

    enumerator kENET_DmaRxWatchdogTimeout
        Rx watchdog timeout.

    enumerator kENET_DmaEarlyTx
        Early transmit.

    enumerator kENET_DmaEarlyRx
        Early receive.

    enumerator kENET_DmaBusErr
        Fatal bus error.

enum _enet_mac_interrupt_enable
    List of mac interrupts supported by the ENET interrupt. This enumeration uses one-hot encoding to allow a logical OR of multiple members.

    Values:

    enumerator kENET_MacPmt

    enumerator kENET_MacTimestamp

enum __enet__event

Defines the common interrupt event for callback use.

Values:

enumerator kENET_RxIntEvent

Receive interrupt event.

enumerator kENET_TxIntEvent

Transmit interrupt event.

enumerator kENET_WakeUpIntEvent

Wake up interrupt event.

enumerator kENET_TimeStampIntEvent

Time stamp interrupt event.

enum __enet__dma__tx__sche

Define the DMA transmit arbitration for multi-queue.

Values:

enumerator kENET_FixPri

Fixed priority. channel 0 has lower priority than channel 1.

enumerator kENET_WeightStrPri

Weighted(burst length) strict priority.

enumerator kENET_WeightRoundRobin

Weighted (weight factor) round robin.

enum __enet__mtl__multiqueue__txsche

Define the MTL Tx scheduling algorithm for multiple queues/rings.

Values:

enumerator kENET_txWeightRR

Tx weight round-robin.

enumerator kENET_txStrPrio

Tx strict priority.

enum __enet__mtl__multiqueue__rxsche

Define the MTL Rx scheduling algorithm for multiple queues/rings.

Values:

enumerator kENET_rxStrPrio

Tx weight round-robin, Rx strict priority.

enumerator kENET_rxWeightStrPrio

Tx strict priority, Rx weight strict priority.

enum __enet__mtl__rxqueuemap

Define the MTL Rx queue and DMA channel mapping.

Values:

enumerator kENET_StaticDirctMap

The received fame in Rx Qn(n = 0,1) direclty map to dma channel n.

enumerator kENET_DynamicMap

The received frame in Rx Qn(n = 0,1) map to the dma channel m(m = 0,1) related with the same Mac.

---

enum __enet_ptp_event_type
> Defines the ENET PTP message related constant.

> *Values:*

> enumerator kENET_PtpEventMsgType
>> PTP event message type.

> enumerator kENET_PtpSrcPortIdLen
>> PTP message sequence id length.

> enumerator kENET_PtpEventPort
>> PTP event port number.

> enumerator kENET_PtpGnrlPort
>> PTP general port number.

enum __enet_tx_offload
> Define the Tx checksum offload options.

> *Values:*

> enumerator kENET_TxOffloadDisable
>> Disable Tx checksum offload.

> enumerator kENET_TxOffloadIPHeader
>> Enable IP header checksum calculation and insertion.

> enumerator kENET_TxOffloadIPHeaderPlusPayload
>> Enable IP header and payload checksum calculation and insertion.

> enumerator kENET_TxOffloadAll
>> Enable IP header, payload and pseudo header checksum calculation and insertion.

typedef enum *_enet_mii_mode* enet_mii_mode_t
> Defines the MII/RMII mode for data interface between the MAC and the PHY.

typedef enum *_enet_mii_speed* enet_mii_speed_t
> Defines the 10/100 Mbps speed for the MII data interface.

typedef enum *_enet_mii_duplex* enet_mii_duplex_t
> Defines the half or full duplex for the MII data interface.

typedef enum *_enet_mii_normal_opcode* enet_mii_normal_opcode_t
> Define the MII opcode for normal MDIO_CLAUSES_22 Frame.

typedef enum *_enet_dma_burstlen* enet_dma_burstlen_t
> Define the DMA maximum transmit burst length.

typedef enum *_enet_desc_flag* enet_desc_flag_t
> Define the flag for the descriptor.

typedef enum *_enet_systime_op* enet_systime_op_t
> Define the system time adjust operation control.

typedef enum *_enet_ts_rollover_type* enet_ts_rollover_type_t
> Define the system time rollover control.

typedef enum *_enet_special_config* enet_special_config_t
> Defines some special configuration for ENET.

> These control flags are provided for special user requirements. Normally, these is no need to set this control flags for ENET initialization. But if you have some special requirements, set the flags to specialControl in the enet_config_t.

Note: "kENET_StoreAndForward" is recommended to be set when the ENET_PTP1588FEATURE_REQUIRED is defined or else the timestamp will be mess-up when the overflow happens.

---

typedef enum *_enet_dma_interrupt_enable* enet_dma_interrupt_enable_t

    List of DMA interrupts supported by the ENET interrupt. This enumeration uses one-hot encoding to allow a logical OR of multiple members.

typedef enum *_enet_mac_interrupt_enable* enet_mac_interrupt_enable_t

    List of mac interrupts supported by the ENET interrupt. This enumeration uses one-hot encoding to allow a logical OR of multiple members.

typedef enum *_enet_event* enet_event_t

    Defines the common interrupt event for callback use.

typedef enum *_enet_dma_tx_sche* enet_dma_tx_sche_t

    Define the DMA transmit arbitration for multi-queue.

typedef enum *_enet_mtl_multiqueue_txsche* enet_mtl_multiqueue_txsche_t

    Define the MTL Tx scheduling algorithm for multiple queues/rings.

typedef enum *_enet_mtl_multiqueue_rxsche* enet_mtl_multiqueue_rxsche_t

    Define the MTL Rx scheduling algorithm for multiple queues/rings.

typedef enum *_enet_mtl_rxqueuemap* enet_mtl_rxqueuemap_t

    Define the MTL Rx queue and DMA channel mapping.

typedef enum *_enet_ptp_event_type* enet_ptp_event_type_t

    Defines the ENET PTP message related constant.

typedef enum *_enet_tx_offload* enet_tx_offload_t

    Define the Tx checksum offload options.

typedef struct *_enet_rx_bd_struct* enet_rx_bd_struct_t

    Defines the receive descriptor structure It has the read-format and write-back format structures. They both have the same size with different region definition. So we define common name as the recive descriptor structure. When initialize the buffer descriptors, read-format region mask bits should be used. When Rx frame has been in the buffer descriptors, write-back format region store the Rx result information.

typedef struct *_enet_tx_bd_struct* enet_tx_bd_struct_t

    Defines the transmit descriptor structure It has the read-format and write-back format structure. They both has the same size with different region definition. So we define common name as the transmit descriptor structure. When initialize the buffer descriptors for Tx, read-format region mask bits should be used. When frame has been transmitted, write-back format region store the Tx result information.

typedef struct *_enet_tx_bd_config_struct* enet_tx_bd_config_struct_t

    Defines the Tx BD configuration structure.

typedef struct *_enet_ptp_time* enet_ptp_time_t

    Defines the ENET PTP time stamp structure.

typedef struct *enet_tx_reclaim_info* enet_tx_reclaim_info_t

    Defines the Tx reclaim information structure.

typedef struct *_enet_tx_dirty_ring* enet_tx_dirty_ring_t

    Defines the ENET transmit dirty addresses ring/queue structure.

---

typedef struct _enet_buffer_config enet_buffer_config_t

Defines the buffer descriptor configure structure.

Notes:

a. The receive and transmit descriptor start address pointer and tail pointer must be word-aligned.

b. The recommended minimum Tx/Rx ring length is 4.

c. The Tx/Rx descriptor tail address shall be the address pointer to the address just after the end of the last last descriptor. because only the descriptors between the start address and the tail address will be used by DMA.

d. The decriptor address is the start address of all used contiguous memory. for example, the rxDescStartAddrAlign is the start address of rxRingLen contiguous descriptor memorise for Rx descriptor ring 0.

e. The "*rxBufferstartAddr" is the first element of rxRingLen (2*rxRingLen for double buffers) Rx buffers. It means the *rxBufferStartAddr is the Rx buffer for the first descriptor the *rxBufferStartAddr + 1 is the Rx buffer for the second descriptor or the Rx buffer for the second buffer in the first descriptor. So please make sure the rxBufferStartAddr is the address of a rxRingLen or 2*rxRingLen array.

typedef struct enet_multiqueue_config enet_multiqueue_config_t

Defines the configuration when multi-queue is used.

typedef void *(*enet_rx_alloc_callback_t)(ENET_Type *base, void *userData, uint8_t channel)

Defines the Rx memory buffer alloc function pointer.

typedef void (*enet_rx_free_callback_t)(ENET_Type *base, void *buffer, void *userData, uint8_t channel)

Defines the Rx memory buffer free function pointer.

typedef struct _enet_config enet_config_t

Defines the basic configuration structure for the ENET device.

Note:

a. Default the signal queue is used so the "multiqueueCfg" is set default with NULL. Set the pointer with a valid configration pointer if the multiple queues are required. If multiple queue is enabled, please make sure the buffer configuration for all are prepared also.

typedef struct _enet_handle enet_handle_t

typedef void (*enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, uint8_t channel, enet_tx_reclaim_info_t *txReclaimInfo, void *userData)

ENET callback function.

typedef struct _enet_tx_bd_ring enet_tx_bd_ring_t

Defines the ENET transmit buffer descriptor ring/queue structure.

typedef struct _enet_rx_bd_ring enet_rx_bd_ring_t

Defines the ENET receive buffer descriptor ring/queue structure.

typedef struct _enet_buffer_struct enet_buffer_struct_t

typedef struct _enet_rx_frame_attribute_struct enet_rx_frame_attribute_t

Rx frame attribute structure.

typedef struct _enet_rx_frame_error enet_rx_frame_error_t

Defines the Rx frame error structure.

typedef struct *_enet_rx_frame_struct* enet_rx_frame_struct_t

    Defines the Rx frame data structure.

typedef struct *_enet_tx_config_struct* enet_tx_config_struct_t

typedef struct *_enet_tx_frame_struct* enet_tx_frame_struct_t

typedef void (*enet_isr_t)(ENET_Type *base, *enet_handle_t* *handle)

const *clock_ip_name_t* s_enetClock[]

    Pointers to enet clocks for each instance.

struct _enet_rx_bd_struct

    #include <fsl_enet.h> Defines the receive descriptor structure It has the read-format and write-back format structures. They both have the same size with different region definition. So we define common name as the recive descriptor structure. When initialize the buffer descriptors, read-format region mask bits should be used. When Rx frame has been in the buffer descriptors, write-back format region store the Rx result information.

    Public Members

    __IO uint32_t rdes0

        Receive descriptor 0

    __IO uint32_t rdes1

        Receive descriptor 1

    __IO uint32_t rdes2

        Receive descriptor 2

    __IO uint32_t rdes3

        Receive descriptor 3

struct _enet_tx_bd_struct

    #include <fsl_enet.h> Defines the transmit descriptor structure It has the read-format and write-back format structure. They both has the same size with different region definition. So we define common name as the transmit descriptor structure. When initialize the buffer descriptors for Tx, read-format region mask bits should be used. When frame has been transmitted, write-back format region store the Tx result information.

    Public Members

    __IO uint32_t tdes0

        Transmit descriptor 0

    __IO uint32_t tdes1

        Transmit descriptor 1

    __IO uint32_t tdes2

        Transmit descriptor 2

    __IO uint32_t tdes3

        Transmit descriptor 3

struct _enet_tx_bd_config_struct

    #include <fsl_enet.h> Defines the Tx BD configuration structure.

Public Members

void *buffer1
    The first buffer address in the descriptor.

uint32_t bytes1
    The bytes in the fist buffer.

void *buffer2
    The second buffer address in the descriptor.

uint32_t bytes2
    The bytes in the second buffer.

uint32_t framelen
    The length of the frame to be transmitted.

bool intEnable
    Interrupt enable flag.

bool tsEnable
    The timestamp enable.

enet_tx_offload_t txOffloadOps
    The Tx checksum offload option, only vaild for Queue 0.

enet_desc_flag_t flag
    The flag of this tx desciriptor, see "enet_qos_desc_flag".

uint8_t slotNum
    The slot number used for AV mode only.

struct __enet__ptp__time
    #include <fsl_enet.h> Defines the ENET PTP time stamp structure.

Public Members

uint64_t second
    Second.

uint32_t nanosecond
    Nanosecond.

struct enet__tx__reclaim__info
    #include <fsl_enet.h> Defines the Tx reclaim information structure.

Public Members

void *context
    User specified data, could be buffer address for free

bool isTsAvail
    Flag indicates timestamp available status

enet_ptp_time_t timeStamp
    Timestamp of frame

struct __enet__tx__dirty__ring
    #include <fsl_enet.h> Defines the ENET transmit dirty addresses ring/queue structure.

Public Members

enet_tx_reclaim_info_t *txDirtyBase
   Dirty buffer descriptor base address pointer.

uint16_t txGenIdx
   Tx generate index.

uint16_t txConsumIdx
   Tx consume index.

uint16_t txRingLen
   Tx ring length.

bool isFull
   Tx ring is full flag, add this parameter to avoid waste one element.

struct __enet__buffer__config
   #include <fsl_enet.h> Defines the buffer descriptor configure structure.

   Notes:

   a. The receive and transmit descriptor start address pointer and tail pointer must be word-aligned.

   b. The recommended minimum Tx/Rx ring length is 4.

   c. The Tx/Rx descriptor tail address shall be the address pointer to the address just after the end of the last last descriptor. because only the descriptors between the start address and the tail address will be used by DMA.

   d. The decriptor address is the start address of all used contiguous memory. for example, the rxDescStartAddrAlign is the start address of rxRingLen contiguous descriptor memorise for Rx descriptor ring 0.

   e. The "*rxBufferstartAddr" is the first element of rxRingLen (2*rxRingLen for double buffers) Rx buffers. It means the *rxBufferStartAddr is the Rx buffer for the first descriptor the *rxBufferStartAddr + 1 is the Rx buffer for the second descriptor or the Rx buffer for the second buffer in the first descriptor. So please make sure the rxBufferStartAddr is the address of a rxRingLen or 2*rxRingLen array.

Public Members

uint8_t rxRingLen
   The length of receive buffer descriptor ring.

uint8_t txRingLen
   The length of transmit buffer descriptor ring.

enet_tx_bd_struct_t *txDescStartAddrAlign
   Aligned transmit descriptor start address.

enet_tx_bd_struct_t *txDescTailAddrAlign
   Aligned transmit descriptor tail address.

enet_tx_reclaim_info_t *txDirtyStartAddr
   Start address of the dirty Tx frame information.

enet_rx_bd_struct_t *rxDescStartAddrAlign
   Aligned receive descriptor start address.

enet_rx_bd_struct_t *rxDescTailAddrAlign
   Aligned receive descriptor tail address.

uint32_t *rxBufferStartAddr

    Start address of the Rx buffers.

uint32_t rxBuffSizeAlign

    Aligned receive data buffer size.

struct enet_multiqueue_config

    #include <fsl_enet.h> Defines the configuration when multi-queue is used.

Public Members

enet_dma_tx_sche_t dmaTxSche

    Transmit arbitation.

enet_dma_burstlen_t burstLen

    Burset len for the queue 1.

uint8_t txdmaChnWeight[(2U)]

    Transmit channel weight.

enet_mtl_multiqueue_txsche_t mtltxSche

    Transmit schedule for multi-queue.

enet_mtl_multiqueue_rxsche_t mtlrxSche

    Receive schedule for multi-queue.

uint8_t rxqueweight[(2U)]

    Refer to the MTL RxQ Control register.

uint32_t txqueweight[(2U)]

    Refer to the MTL TxQ Quantum Weight register.

uint8_t rxqueuePrio[(2U)]

    Receive queue priority.

uint8_t txqueuePrio[(2U)]

    Refer to Transmit Queue Priority Mapping register.

enet_mtl_rxqueuemap_t mtlrxQuemap

    Rx queue DMA Channel mapping.

struct _enet_config

    #include <fsl_enet.h> Defines the basic configuration structure for the ENET device.

    Note:

    a. Default the signal queue is used so the "multiqueueCfg" is set default with NULL. Set the pointer with a valid configration pointer if the multiple queues are required. If multiple queue is enabled, please make sure the buffer configuration for all are prepared also.

Public Members

uint16_t specialControl

    The logicl or of enet_special_config_t

enet_multiqueue_config_t *multiqueueCfg

    Use both Tx/Rx queue(dma channel) 0 and 1.

uint32_t interrupt

MAC interrupt source. A logical OR of enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t.

enet_mii_mode_t miiMode

MII mode.

enet_mii_speed_t miiSpeed

MII Speed.

enet_mii_duplex_t miiDuplex

MII duplex.

uint16_t pauseDuration

Used in the Tx flow control frame, only valid when kENET_FlowControlEnable is set.

enet_rx_alloc_callback_t rxBuffAlloc

Callback to alloc memory, must be provided for zero-copy Rx.

enet_rx_free_callback_t rxBuffFree

Callback to free memory, must be provided for zero-copy Rx.

struct __enet__tx__bd__ring

#include <fsl_enet.h> Defines the ENET transmit buffer descriptor ring/queue structure.

Public Members

enet_tx_bd_struct_t *txBdBase

Buffer descriptor base address pointer.

uint16_t txGenIdx

Tx generate index.

uint16_t txConsumIdx

Tx consum index.

volatile uint16_t txDescUsed

Tx descriptor used number.

uint16_t txRingLen

Tx ring length.

struct __enet__rx__bd__ring

#include <fsl_enet.h> Defines the ENET receive buffer descriptor ring/queue structure.

Public Members

enet_rx_bd_struct_t *rxBdBase

Buffer descriptor base address pointer.

uint16_t rxGenIdx

The current available receive buffer descriptor pointer.

uint16_t rxRingLen

Receive ring length.

uint32_t rxBuffSizeAlign

Receive buffer size.

struct __enet__handle

#include <fsl_enet.h> Defines the ENET handler structure.

2.26. ENET: Ethernet Driver

Public Members

bool multiQueEnable
    Multi-queue enable status.

bool doubleBuffEnable
    The double buffer enable status.

bool rxintEnable
    Rx interrupt enable status.

enet_rx_bd_ring_t rxBdRing[(2U)]
    Receive buffer descriptor.

enet_tx_bd_ring_t txBdRing[(2U)]
    Transmit buffer descriptor.

enet_tx_dirty_ring_t txDirtyRing[(2U)]
    Transmit dirty buffers addresses.

uint32_t *rxBufferStartAddr[(2U)]
    The Init-Rx buffers used for reinit corrupted BD due to write-back operation.

uint32_t txLenLimitation[(2U)]
    Tx frame length limitation.

enet_callback_t callback
    Callback function.

void *userData
    Callback function parameter.

enet_rx_alloc_callback_t rxBuffAlloc
    Callback to alloc memory, must be provided for zero-copy Rx.

enet_rx_free_callback_t rxBuffFree
    Callback to free memory, must be provided for zero-copy Rx.

struct __enet__buffer__struct
    #include <fsl_enet.h>

Public Members

void *buffer
    The buffer stores the whole or partial frame.

uint16_t length
    The byte length of this buffer.

struct __enet__rx__frame__attribute__struct
    #include <fsl_enet.h> Rx frame attribute structure.

Public Members

bool isTsAvail
    Rx frame timestamp is available or not.

enet_ptp_time_t timestamp
    The nanosecond part timestamp of this Rx frame.

struct __enet__rx__frame__error
    #include <fsl_enet.h> Defines the Rx frame error structure.

Public Members

bool statsDribbleErr

The received packet has a non-integer multiple of bytes (odd nibbles).

bool statsRxErr

Receive error.

bool statsOverflowErr

Rx FIFO overflow error.

bool statsWatchdogTimeoutErr

Receive watchdog timeout.

bool statsGaintPacketErr

Receive error.

bool statsRxFcsErr

Receive CRC error.

struct __enet__rx__frame__struct

#include <fsl_enet.h> Defines the Rx frame data structure.

Public Members

enet_buffer_struct_t *rxBuffArray

Rx frame buffer structure.

uint16_t totLen

Rx frame total length.

enet_rx_frame_attribute_t rxAttribute

Rx frame attribute structure.

enet_rx_frame_error_t rxFrameError

Rx frame error.

struct __enet__tx__config__struct

#include <fsl_enet.h>

Public Members

uint8_t intEnable

Enable interrupt every time one BD is completed.

uint8_t tsEnable

Transmit timestamp enable.

uint8_t slotNum

Slot number control bits in AV mode.

enet_tx_offload_t txOffloadOps

Tx checksum offload option.

struct __enet__tx__frame__struct

#include <fsl_enet.h>

Public Members

enet_buffer_struct_t *txBuffArray
    Tx frame buffer structure.

uint32_t txBuffNum
    Buffer number of this Tx frame.

enet_tx_config_struct_t txConfig
    Tx extra configuation.

void *context
    Driver reclaims and gives it in Tx over callback.

2.27 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

2.28 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.

2.29 LCDC: LCD Controller Driver

status_t LCDC_Init(LCD_Type *base, const *lcdc_config_t* *config, uint32_t srcClock_Hz)
Initialize the LCD module.

Parameters

- base – LCD peripheral base address.

- config – Pointer to configuration structure, see to lcdc_config_t.

- srcClock_Hz – The LCD input clock (LCDCLK) frequency in Hz.

Return values

- kStatus_Success – LCD is initialized successfully.

- kStatus_InvalidArgument – Initlialize failed because of invalid argument.

void LCDC_Deinit(LCD_Type *base)
Deinitialize the LCD module.

Parameters

- base – LCD peripheral base address.

void LCDC_GetDefaultConfig(*lcdc_config_t* *config)

Gets default pre-defined settings for initial configuration.

This function initializes the configuration structure. The default values are:

```
config->panelClock_Hz = 0U;
config->ppl = 0U;
config->hsw = 0U;
config->hfp = 0U;
config->hbp = 0U;
config->lpp = 0U;
config->vsw = 0U;
config->vfp = 0U;
config->vbp = 0U;
config->acBiasFreq = 1U;
config->polarityFlags = 0U;
config->enableLineEnd = false;
config->lineEndDelay = 0U;
config->upperPanelAddr = 0U;
config->lowerPanelAddr = 0U;
config->bpp = kLCDC_1BPP;
config->dataFormat = kLCDC_LittleEndian;
config->swapRedBlue = false;
config->display = kLCDC_DisplayTFT;
```

Parameters

- config – Pointer to configuration structure.

static inline void LCDC_Start(LCD_Type *base)

Start to output LCD timing signal.

The LCD power up sequence should be:

a. Apply power to LCD, here all output signals are held low.

b. When LCD power stablized, call LCDC_Start to output the timing signals.

c. Apply contrast voltage to LCD panel. Delay if the display requires.

d. Call LCDC_PowerUp.

Parameters

- base – LCD peripheral base address.

static inline void LCDC_Stop(LCD_Type *base)

Stop the LCD timing signal.

The LCD power down sequence should be:

a. Call LCDC_PowerDown.

b. Delay if the display requires. Disable contrast voltage to LCD panel.

c. Call LCDC_Stop to disable the timing signals.

d. Disable power to LCD.

Parameters

- base – LCD peripheral base address.

static inline void LCDC_PowerUp(LCD_Type *base)

> Power up the LCD and output the pixel signal.

> > **Parameters**

> > > • base – LCD peripheral base address.

static inline void LCDC_PowerDown(LCD_Type *base)

> Power down the LCD and disable the output pixel signal.

> > **Parameters**

> > > • base – LCD peripheral base address.

void LCDC_SetPanelAddr(LCD_Type *base, *lcdc_panel_t* panel, uint32_t addr)

> Sets panel frame base address.

> > **Parameters**

> > > • base – LCD peripheral base address.
> > >
> > > • panel – Which panel to set.
> > >
> > > • addr – Frame base address, must be doubleword(64-bit) aligned.

void LCDC_SetPalette(LCD_Type *base, const uint32_t *palette, uint8_t count_words)

> Sets palette.

> > **Parameters**

> > > • base – LCD peripheral base address.
> > >
> > > • palette – Pointer to the palette array.
> > >
> > > • count_words – Length of the palette array to set (how many words), it should not be larger than LCDC_PALETTE_SIZE_WORDS.

static inline void LCDC_SetVerticalInterruptMode(LCD_Type *base, *lcdc_vertical_compare_interrupt_mode_t* mode)

> Sets the vertical compare interrupt mode.

> > **Parameters**

> > > • base – LCD peripheral base address.
> > >
> > > • mode – The vertical compare interrupt mode.

void LCDC_EnableInterrupts(LCD_Type *base, uint32_t mask)

> Enable LCD interrupts.

> Example to enable LCD base address update interrupt and vertical compare interrupt:

```
LCDC_EnableInterrupts(LCD, kLCDC_BaseAddrUpdateInterrupt | kLCDC_
↪VerticalCompareInterrupt);
```

> > **Parameters**

> > > • base – LCD peripheral base address.
> > >
> > > • mask – Interrupts to enable, it is OR'ed value of _lcdc_interrupts.

void LCDC_DisableInterrupts(LCD_Type *base, uint32_t mask)

> Disable LCD interrupts.

> Example to disable LCD base address update interrupt and vertical compare interrupt:

```
LCDC_DisableInterrupts(LCD, kLCDC_BaseAddrUpdateInterrupt | kLCDC_
↪VerticalCompareInterrupt);
```

Parameters

- base – LCD peripheral base address.

- mask – Interrupts to disable, it is OR'ed value of _lcdc_interrupts.

uint32_t LCDC_GetInterruptsPendingStatus(LCD_Type *base)

Get LCD interrupt pending status.

Example:

```
uint32_t status;

status = LCDC_GetInterruptsPendingStatus(LCD);

if (kLCDC_BaseAddrUpdateInterrupt & status)
{
    LCD base address update interrupt occurred.
}

if (kLCDC_VerticalCompareInterrupt & status)
{
    LCD vertical compare interrupt occurred.
}
```

Parameters

- base – LCD peripheral base address.

Returns

Interrupts pending status, it is OR'ed value of _lcdc_interrupts.

uint32_t LCDC_GetEnabledInterruptsPendingStatus(LCD_Type *base)

Get LCD enabled interrupt pending status.

This function is similar with LCDC_GetInterruptsPendingStatus, the only difference is, this function only returns the pending status of the interrupts that have been enabled using LCDC_EnableInterrupts.

Parameters

- base – LCD peripheral base address.

Returns

Interrupts pending status, it is OR'ed value of _lcdc_interrupts.

void LCDC_ClearInterruptsStatus(LCD_Type *base, uint32_t mask)

Clear LCD interrupts pending status.

Example to clear LCD base address update interrupt and vertical compare interrupt pending status:

```
LCDC_ClearInterruptsStatus(LCD, kLCDC_BaseAddrUpdateInterrupt | kLCDC_
↪VerticalCompareInterrupt);
```

Parameters

- base – LCD peripheral base address.

- mask – Interrupts to disable, it is OR'ed value of _lcdc_interrupts.

void LCDC_SetCursorConfig(LCD_Type *base, const *lcdc_cursor_config_t* *config)

Set the hardware cursor configuration.

This function should be called before enabling the hardware cursor. It supports initializing multiple cursor images at a time when using 32x32 pixels cursor.

For example:

```
uint32_t cursor0Img[LCDC_CURSOR_IMG_32X32_WORDS] = {...};
uint32_t cursor2Img[LCDC_CURSOR_IMG_32X32_WORDS] = {...};

lcdc_cursor_config_t cursorConfig;

LCDC_CursorGetDefaultConfig(&cursorConfig);

cursorConfig.image[0] = cursor0Img;
cursorConfig.image[2] = cursor2Img;

LCDC_SetCursorConfig(LCD, &cursorConfig);

LCDC_ChooseCursor(LCD, 0);
LCDC_SetCursorPosition(LCD, 0, 0);

LCDC_EnableCursor(LCD);
```

In this example, cursor 0 and cursor 2 image data are initialized, but cursor 1 and cursor 3 image data are not initialized because image[1] and image[2] are all NULL. With this, application could initializes all cursor images it will use at the beginning and call LCDC_SetCursorImage directly to display the one which it needs.

> **Parameters**
> - base – LCD peripheral base address.
> - config – Pointer to the hardware cursor configuration structure.

void LCDC_CursorGetDefaultConfig(*lcdc_cursor_config_t* *config)

Get the hardware cursor default configuration.

The default configuration values are:

```
config->size = kLCDC_CursorSize32;
config->syncMode = kLCDC_CursorAsync;
config->palette0.red = 0U;
config->palette0.green = 0U;
config->palette0.blue = 0U;
config->palette1.red = 255U;
config->palette1.green = 255U;
config->palette1.blue = 255U;
config->image[0] = (uint32_t *)0;
config->image[1] = (uint32_t *)0;
config->image[2] = (uint32_t *)0;
config->image[3] = (uint32_t *)0;
```

> **Parameters**
> - config – Pointer to the hardware cursor configuration structure.

static inline void LCDC_EnableCursor(LCD_Type *base, bool enable)

Enable or disable the cursor.

> **Parameters**
> - base – LCD peripheral base address.
> - enable – True to enable, false to disable.

static inline void LCDC_ChooseCursor(LCD_Type *base, uint8_t index)

Choose which cursor to display.

When using 32x32 cursor, the number of cursors supports is LCDC_CURSOR_COUNT. When using 64x64 cursor, the LCD only supports one cursor. This function selects which cursor

to display when using 32x32 cursor. When synchronization mode is kLCDC_CursorSync, the change effects in the next frame. When synchronization mode is kLCDC_CursorAsync, change effects immediately.

Note: The function LCDC_SetCursorPosition must be called after this function to show the new cursor.

> **Parameters**
>
> - base – LCD peripheral base address.
> - index – Index of the cursor to display.

void LCDC_SetCursorPosition(LCD_Type *base, int32_t positionX, int32_t positionY)

Set the position of cursor.

When synchronization mode is kLCDC_CursorSync, position change effects in the next frame. When synchronization mode is kLCDC_CursorAsync, position change effects immediately.

> **Parameters**
>
> - base – LCD peripheral base address.
> - positionX – X ordinate of the cursor top-left measured in pixels
> - positionY – Y ordinate of the cursor top-left measured in pixels

void LCDC_SetCursorImage(LCD_Type *base, *lcdc_cursor_size_t* size, uint8_t index, const uint32_t *image)

Set the cursor image.

The interrupt kLCDC_CursorInterrupt indicates that last cursor pixel is displayed. When the hardware cursor is enabled,

> **Parameters**
>
> - base – LCD peripheral base address.
> - size – The cursor size.
> - index – Index of the cursor to set when using 32x32 cursor.
> - image – Pointer to the cursor image. When using 32x32 cursor, the image size should be LCDC_CURSOR_IMG_32X32_WORDS. When using 64x64 cursor, the image size should be LCDC_CURSOR_IMG_64X64_WORDS.

FSL_LCDC_DRIVER_VERSION

LCDC driver version.

enum __lcdc_polarity_flags

LCD sigal polarity flags.

Values:

enumerator kLCDC_InvertVsyncPolarity

Invert the VSYNC polarity, set to active low.

enumerator kLCDC_InvertHsyncPolarity

Invert the HSYNC polarity, set to active low.

enumerator kLCDC_InvertClkPolarity

Invert the panel clock polarity, set to drive data on falling edge.

enumerator kLCDC_InvertDePolarity
        Invert the data enable (DE) polarity, set to active low.

enum __lcdc_bpp
    LCD bits per pixel.

    Values:

    enumerator kLCDC_1BPP
        1 bpp.

    enumerator kLCDC_2BPP
        2 bpp.

    enumerator kLCDC_4BPP
        4 bpp.

    enumerator kLCDC_8BPP
        8 bpp.

    enumerator kLCDC_16BPP
        16 bpp.

    enumerator kLCDC_24BPP
        24 bpp, TFT panel only.

    enumerator kLCDC_16BPP565
        16 bpp, 5:6:5 mode.

    enumerator kLCDC_12BPP
        12 bpp, 4:4:4 mode.

enum __lcdc_display
    The types of display panel.

    Values:

    enumerator kLCDC_DisplayTFT
        Active matrix TFT panels with up to 24-bit bus interface.

    enumerator kLCDC_DisplaySingleMonoSTN4Bit
        Single-panel monochrome STN (4-bit bus interface).

    enumerator kLCDC_DisplaySingleMonoSTN8Bit
        Single-panel monochrome STN (8-bit bus interface).

    enumerator kLCDC_DisplayDualMonoSTN4Bit
        Dual-panel monochrome STN (4-bit bus interface).

    enumerator kLCDC_DisplayDualMonoSTN8Bit
        Dual-panel monochrome STN (8-bit bus interface).

    enumerator kLCDC_DisplaySingleColorSTN8Bit
        Single-panel color STN (8-bit bus interface).

    enumerator kLCDC_DisplayDualColorSTN8Bit
        Dual-panel coor STN (8-bit bus interface).

enum __lcdc_data_format
    LCD panel buffer data format.

    Values:

enumerator kLCDC_LittleEndian
    Little endian byte, little endian pixel.

enumerator kLCDC_BigEndian
    Big endian byte, big endian pixel.

enumerator kLCDC_WinCeMode
    little-endian byte, big-endian pixel for Windows CE mode.

enum _lcdc_vertical_compare_interrupt_mode
    LCD vertical compare interrupt mode.

    Values:

    enumerator kLCDC_StartOfVsync
        Generate vertical compare interrupt at start of VSYNC.

    enumerator kLCDC_StartOfBackPorch
        Generate vertical compare interrupt at start of back porch.

    enumerator kLCDC_StartOfActiveVideo
        Generate vertical compare interrupt at start of active video.

    enumerator kLCDC_StartOfFrontPorch
        Generate vertical compare interrupt at start of front porch.

enum _lcdc_interrupts
    LCD interrupts.

    Values:

    enumerator kLCDC_CursorInterrupt
        Cursor image read finished interrupt.

    enumerator kLCDC_FifoUnderflowInterrupt
        FIFO underflow interrupt.

    enumerator kLCDC_BaseAddrUpdateInterrupt
        Panel frame base address update interrupt.

    enumerator kLCDC_VerticalCompareInterrupt
        Vertical compare interrupt.

    enumerator kLCDC_AhbErrorInterrupt
        AHB master error interrupt.

enum _lcdc_panel
    LCD panel frame.

    Values:

    enumerator kLCDC_UpperPanel
        Upper panel frame.

    enumerator kLCDC_LowerPanel
        Lower panel frame.

enum _lcdc_cursor_size
    LCD hardware cursor size.

    Values:

    enumerator kLCDC_CursorSize32
        32x32 pixel cursor.

enumerator kLCDC_CursorSize64
64x64 pixel cursor.

enum _lcdc_cursor_sync_mode
LCD hardware cursor frame synchronization mode.

Values:

enumerator kLCDC_CursorAsync
Cursor change will be displayed immediately.

enumerator kLCDC_CursorSync
Cursor change will be displayed in next frame.

typedef enum *_lcdc_bpp* lcdc_bpp_t
LCD bits per pixel.

typedef enum *_lcdc_display* lcdc_display_t
The types of display panel.

typedef enum *_lcdc_data_format* lcdc_data_format_t
LCD panel buffer data format.

typedef struct *_lcdc_config* lcdc_config_t
LCD configuration structure.

typedef enum *_lcdc_vertical_compare_interrupt_mode* lcdc_vertical_compare_interrupt_mode_t
LCD vertical compare interrupt mode.

typedef enum *_lcdc_panel* lcdc_panel_t
LCD panel frame.

typedef enum *_lcdc_cursor_size* lcdc_cursor_size_t
LCD hardware cursor size.

typedef struct *_lcdc_cursor_palette* lcdc_cursor_palette_t
LCD hardware cursor palette.

typedef enum *_lcdc_cursor_sync_mode* lcdc_cursor_sync_mode_t
LCD hardware cursor frame synchronization mode.

typedef struct *_lcdc_cursor_config* lcdc_cursor_config_t
LCD hardware cursor configuration structure.

LCDC_CURSOR_COUNT
How many hardware cursors supports.

LCDC_CURSOR_IMG_BPP
LCD cursor image bits per pixel.

LCDC_CURSOR_IMG_32X32_WORDS
LCD 32x32 cursor image size in word(32-bit).

LCDC_CURSOR_IMG_64X64_WORDS
LCD 64x64 cursor image size in word(32-bit).

LCDC_PALETTE_SIZE_WORDS
LCD palette size in words(32-bit).

struct _lcdc_config
#include <fsl_lcdc.h> LCD configuration structure.

Public Members

uint32_t panelClock_Hz
    Panel clock in Hz.

uint16_t ppl
    Pixels per line, it must could be divided by 16.

uint8_t hsw
    HSYNC pulse width.

uint8_t hfp
    Horizontal front porch.

uint8_t hbp
    Horizontal back porch.

uint16_t lpp
    Lines per panal.

uint8_t vsw
    VSYNC pulse width.

uint8_t vfp
    Vrtical front porch.

uint8_t vbp
    Vertical back porch.

uint8_t acBiasFreq
    The number of line clocks between AC bias pin toggling. Only used for STN display.

uint16_t polarityFlags
    OR'ed value of _lcdc_polarity_flags, used to contol the signal polarity.

bool enableLineEnd
    Enable line end or not, the line end is a positive pulse with 4 panel clock.

uint8_t lineEndDelay
    The panel clocks between the last pixel of line and the start of line end.

uint32_t upperPanelAddr
    LCD upper panel base address, must be double-word(64-bit) align.

uint32_t lowerPanelAddr
    LCD lower panel base address, must be double-word(64-bit) align.

lcdc_bpp_t bpp
    LCD bits per pixel.

lcdc_data_format_t dataFormat
    Data format.

bool swapRedBlue
    Set true to use BGR format, set false to choose RGB format.

lcdc_display_t display
    The display type.

struct __lcdc_cursor_palette
    #include <fsl_lcdc.h> LCD hardware cursor palette.

Public Members

uint8_t red
    Red color component.

uint8_t green
    Red color component.

uint8_t blue
    Red color component.

struct __lcdc_cursor_config
    #include <fsl_lcdc.h> LCD hardware cursor configuration structure.

Public Members

lcdc_cursor_size_t size
    Cursor size.

lcdc_cursor_sync_mode_t syncMode
    Cursor synchronization mode.

lcdc_cursor_palette_t palette0
    Cursor palette 0.

lcdc_cursor_palette_t palette1
    Cursor palette 1.

uint32_t *image[4U]
    Pointer to cursor image data.

2.30 MCAN: Controller Area Network Driver

void MCAN_Init(CAN_Type *base, const *mcan_config_t* *config, uint32_t sourceClock_Hz)
    Initializes an MCAN instance.

    This function initializes the MCAN module with user-defined settings. This example shows how to set up the mcan_config_t parameters and how to call the MCAN_Init function by passing in these parameters.

```
mcan_config_t config;
config->baudRateA = 500000U;
config->baudRateD = 1000000U;
config->enableCanfdNormal = false;
config->enableCanfdSwitch = false;
config->enableLoopBackInt = false;
config->enableLoopBackExt = false;
config->enableBusMon = false;
MCAN_Init(CANFD0, &config, 8000000UL);
```

Parameters

- base – MCAN peripheral base address.
- config – Pointer to the user-defined configuration structure.
- sourceClock_Hz – MCAN Protocol Engine clock source frequency in Hz.

void MCAN_Deinit(CAN_Type *base)

> Deinitializes an MCAN instance.

> This function deinitializes the MCAN module.

>> **Parameters**

>>> • base – MCAN peripheral base address.

void MCAN_GetDefaultConfig(*mcan_config_t* *config)

> Gets the default configuration structure.

> This function initializes the MCAN configuration structure to default values. The default values are as follows. config->baudRateA = 500000U; config->baudRateD = 1000000U; config->enableCanfdNormal = false; config->enableCanfdSwitch = false; config->enableLoopBackInt = false; config->enableLoopBackExt = false; config->enableBusMon = false;

>> **Parameters**

>>> • config – Pointer to the MCAN configuration structure.

static inline void MCAN_EnterInitialMode(CAN_Type *base)

> MCAN enters initialization mode.

> After enter initialization mode, users can write access to the protected configuration registers.

>> **Parameters**

>>> • base – MCAN peripheral base address.

static inline void MCAN_EnterNormalMode(CAN_Type *base)

> MCAN enters normal mode.

> After initialization, INIT bit in CCCR register must be cleared to enter normal mode thus synchronizes to the CAN bus and ready for communication.

>> **Parameters**

>>> • base – MCAN peripheral base address.

static inline void MCAN_SetMsgRAMBase(CAN_Type *base, uint32_t value)

> Sets the MCAN Message RAM base address.

> This function sets the Message RAM base address.

>> **Parameters**

>>> • base – MCAN peripheral base address.

>>> • value – Desired Message RAM base.

static inline uint32_t MCAN_GetMsgRAMBase(CAN_Type *base)

> Gets the MCAN Message RAM base address.

> This function gets the Message RAM base address.

>> **Parameters**

>>> • base – MCAN peripheral base address.

>> **Returns**

>>> Message RAM base address.

bool MCAN_CalculateImprovedTimingValues(uint32_t baudRate, uint32_t sourceClock_Hz, *mcan_timing_config_t* *pconfig)

> Calculates the improved timing values by specific baudrates for classical CAN.

>> **Parameters**

- baudRate – The classical CAN speed in bps defined by user

- sourceClock_Hz – The Source clock data speed in bps. Zero to disable baudrate switching

- pconfig – Pointer to the MCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

bool MCAN_CalculateSpecifiedTimingValues(uint32_t sourceClock_Hz, *mcan_timing_config_t* *pconfig, const *mcan_timing_param_t* *pParamConfig)

Calculates the specified timing values for classical CAN with user-defined settings.

User can specify baudrates, sample point position, bus length, and transceiver propagation delay. This example shows how to set up the mcan_timing_param_t parameters and how to call the this function by passing in these parameters.

```
mcan_timing_config_t timing_config;
mcan_timing_param_t timing_param;
timing_param.busLength = 1U;
timing_param.propTxRx = 230U;
timing_param.nominalbaudRate = 500000U;
timing_param.nominalSP = 800U;
MCAN_CalculateSpecifiedTimingValues(MCAN_CLK_FREQ, &timing_config, &timing_param);
```

Note that due to integer division will sacrifice the precision, actual sample point may not equal to expected. If actual sample point is not in allowed 2% range, this function will return false. So it is better to select higher source clock when baudrate is relatively high. This will ensure more time quanta and higher precision of sample point. Parameter busLength and propTxRx are optional and intended to verify whether propagation delay is too long to corrupt sample point. User can set these parameter zero if you do not want to consider this factor.

Parameters

- sourceClock_Hz – The Source clock data speed in bps.

- pconfig – Pointer to the MCAN timing configuration structure.

- config – Pointer to the MCAN timing parameters structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

void MCAN_SetArbitrationTimingConfig(CAN_Type *base, const *mcan_timing_config_t* *config)

Sets the MCAN protocol arbitration phase timing characteristic.

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the MCAN_Init() and fill the baud rate field with a desired value. This provides the default arbitration phase timing characteristics.

Note that calling MCAN_SetArbitrationTimingConfig() overrides the baud rate set in MCAN_Init().

Parameters

- base – MCAN peripheral base address.

- config – Pointer to the timing configuration structure.

status_t MCAN_SetBaudRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t baudRate_Bps)

Set Baud Rate of MCAN classic mode.

This function set the baud rate of MCAN base on MCAN_CalculateImprovedTimingValues() API calculated timing values.

Parameters

- base – MCAN peripheral base address.

- sourceClock_Hz – Source Clock in Hz.

- baudRate_Bps – Baud Rate in Bps.

Returns

kStatus_Success - Set CAN baud rate (only has Nominal phase) successfully.

bool MCAN_FDCalculateImprovedTimingValues(uint32_t baudRate, uint32_t baudRateFD, uint32_t sourceClock_Hz, *mcan_timing_config_t* *pconfig)

Calculates the improved timing values by specific baudrates for CANFD.

Parameters

- baudRate – The CANFD bus control speed in bps defined by user

- baudRateFD – The CANFD bus data speed in bps defined by user

- sourceClock_Hz – The Source clock data speed in bps.

- pconfig – Pointer to the MCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

bool MCAN_FDCalculateSpecifiedTimingValues(uint32_t sourceClock_Hz, *mcan_timing_config_t* *pconfig, const *mcan_timing_param_t* *pParamConfig)

Calculates the specified timing values for CANFD with user-defined settings.

User can specify baudrates, sample point position, bus length, and transceiver propagation delay. This example shows how to set up the mcan_timing_param_t parameters and how to call the this function by passing in these parameters.

```
mcan_timing_config_t timing_config;
mcan_timing_param_t timing_param;
timing_param.busLength = 1U;
timing_param.propTxRx = 230U;
timing_param.nominalbaudRate = 500000U;
timing_param.nominalSP = 800U;
timing_param.databaudRate = 4000000U;
timing_param.dataSP = 700U;
MCAN_FDCalculateSpecifiedTimingValues(MCAN_CLK_FREQ, &timing_config, &timing_param);
```

Note that due to integer division will sacrifice the precision, actual sample point may not equal to expected. So it is better to select higher source clock when baudrate is relatively high. Select higher nominal baudrate when source clock is relatively high because large clock predivider will lead to less time quanta in data phase. This function will set predivider in arbitration phase equal to data phase. These methods will ensure more time quanta and higher precision of sample point. Parameter busLength and propTxRx are optional and intended to verify whether propagation delay is too long to corrupt sample point. User can set these parameter zero if you do not want to consider this factor.

Parameters

- sourceClock_Hz – The Source clock data speed in bps.

- pconfig – Pointer to the MCAN timing configuration structure.

- config – Pointer to the MCAN timing parameters structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

status_t MCAN_SetBaudRateFD(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t baudRateN_Bps, uint32_t baudRateD_Bps)

Set Baud Rate of MCAN FD mode.

This function set the baud rate of MCAN FD base on MCAN_FDCalculateImprovedTimingValues API calculated timing values.

Parameters

- base – MCAN peripheral base address.

- sourceClock_Hz – Source Clock in Hz.

- baudRateN_Bps – Nominal Baud Rate in Bps.

- baudRateD_Bps – Data Baud Rate in Bps.

Returns

kStatus_Success - Set CAN FD baud rate (include Nominal and Data phase) successfully.

void MCAN_SetDataTimingConfig(CAN_Type *base, const *mcan_timing_config_t* *config)

Sets the MCAN protocol data phase timing characteristic.

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the MCAN_Init() and fill the baud rate field with a desired value. This provides the default data phase timing characteristics.

Note that calling MCAN_SetArbitrationTimingConfig() overrides the baud rate set in MCAN_Init().

Parameters

- base – MCAN peripheral base address.

- config – Pointer to the timing configuration structure.

void MCAN_SetRxFifo0Config(CAN_Type *base, const *mcan_rx_fifo_config_t* *config)

Configures an MCAN receive fifo 0 buffer.

This function sets start address, element size, watermark, operation mode and datafield size of the recieve fifo 0.

Parameters

- base – MCAN peripheral base address.

- config – The receive fifo 0 configuration structure.

void MCAN_SetRxFifo1Config(CAN_Type *base, const *mcan_rx_fifo_config_t* *config)

Configures an MCAN receive fifo 1 buffer.

This function sets start address, element size, watermark, operation mode and datafield size of the recieve fifo 1.

Parameters

- base – MCAN peripheral base address.

- config – The receive fifo 1 configuration structure.

void MCAN_SetRxBufferConfig(CAN_Type *base, const *mcan_rx_buffer_config_t* *config)

Configures an MCAN receive buffer.

This function sets start address and datafield size of the recieve buffer.

Parameters

- base – MCAN peripheral base address.

- config – The receive buffer configuration structure.

void MCAN_SetTxEventFifoConfig(CAN_Type *base, const *mcan_tx_fifo_config_t* *config)

Configures an MCAN transmit event fifo.

This function sets start address, element size, watermark of the transmit event fifo.

Parameters

- base – MCAN peripheral base address.

- config – The transmit event fifo configuration structure.

void MCAN_SetTxBufferConfig(CAN_Type *base, const *mcan_tx_buffer_config_t* *config)

Configures an MCAN transmit buffer.

This function sets start address, element size, fifo/queue mode and datafield size of the transmit buffer.

Parameters

- base – MCAN peripheral base address.

- config – The transmit buffer configuration structure.

void MCAN_SetFilterConfig(CAN_Type *base, const *mcan_frame_filter_config_t* *config)

Set filter configuration.

This function sets remote and non masking frames in global filter configuration, also the start address, list size in standard/extended ID filter configuration.

Parameters

- base – MCAN peripheral base address.

- config – The MCAN filter configuration.

status_t MCAN_SetMessageRamConfig(CAN_Type *base, const *mcan_memory_config_t* *config)

Set Message RAM related configuration.

Note: This function include Standard/extended ID filter, Rx FIFO 0/1, Rx buffer, Tx event FIFO and Tx buffer configurations

---

Parameters

- base – MCAN peripheral base address.

- config – The MCAN filter configuration.

Return values

- kStatus_Success – - Message RAM related configuration Successfully.

- kStatus_Fail – - Message RAM related configure fail due to wrong address parameter.

void MCAN_SetSTDFilterElement(CAN_Type *base, const *mcan_frame_filter_config_t* *config, const *mcan_std_filter_element_config_t* *filter, uint8_t idx)

Set standard message ID filter element configuration.

Parameters

- base – MCAN peripheral base address.

- config – The MCAN filter configuration.

- filter – The MCAN standard message ID filter element configuration.

- idx – The standard message ID filter element index.

void MCAN_SetEXTFilterElement(CAN_Type *base, const *mcan_frame_filter_config_t* *config,
const *mcan_ext_filter_element_config_t* *filter, uint8_t idx)

Set extended message ID filter element configuration.

Parameters

- base – MCAN peripheral base address.
- config – The MCAN filter configuration.
- filter – The MCAN extended message ID filter element configuration.
- idx – The extended message ID filter element index.

static inline uint32_t MCAN_GetStatusFlag(CAN_Type *base, uint32_t mask)

Gets the MCAN module interrupt flags.

This function gets all MCAN interrupt status flags.

Parameters

- base – MCAN peripheral base address.
- mask – The ORed MCAN interrupt mask.

Returns

MCAN status flags which are ORed.

static inline void MCAN_ClearStatusFlag(CAN_Type *base, uint32_t mask)

Clears the MCAN module interrupt flags.

This function clears MCAN interrupt status flags.

Parameters

- base – MCAN peripheral base address.
- mask – The ORed MCAN interrupt mask.

static inline bool MCAN_GetRxBufferStatusFlag(CAN_Type *base, uint8_t idx)

Gets the new data flag of specific Rx Buffer.

This function gets new data flag of specific Rx Buffer.

Parameters

- base – MCAN peripheral base address.
- idx – Rx Buffer index.

Returns

Rx Buffer new data status flag.

static inline void MCAN_ClearRxBufferStatusFlag(CAN_Type *base, uint8_t idx)

Clears the new data flag of specific Rx Buffer.

This function clears new data flag of specific Rx Buffer.

Parameters

- base – MCAN peripheral base address.
- idx – Rx Buffer index.

static inline void MCAN_EnableInterrupts(CAN_Type *base, uint32_t line, uint32_t mask)

Enables MCAN interrupts according to the provided interrupt line and mask.

This function enables the MCAN interrupts according to the provided interrupt line and mask. The mask is a logical OR of enumeration members.

2.30. MCAN: Controller Area Network Driver

Parameters

- base – MCAN peripheral base address.

- line – Interrupt line number, 0 or 1.

- mask – The interrupts to enable.

static inline void MCAN_EnableTransmitBufferInterrupts(CAN_Type *base, uint8_t idx)

Enables MCAN Tx Buffer interrupts according to the provided index.

This function enables the MCAN Tx Buffer interrupts.

Parameters

- base – MCAN peripheral base address.

- idx – Tx Buffer index.

static inline void MCAN_DisableTransmitBufferInterrupts(CAN_Type *base, uint8_t idx)

Disables MCAN Tx Buffer interrupts according to the provided index.

This function disables the MCAN Tx Buffer interrupts.

Parameters

- base – MCAN peripheral base address.

- idx – Tx Buffer index.

static inline void MCAN_DisableInterrupts(CAN_Type *base, uint32_t mask)

Disables MCAN interrupts according to the provided mask.

This function disables the MCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members.

Parameters

- base – MCAN peripheral base address.

- mask – The interrupts to disable.

uint32_t MCAN_IsTransmitRequestPending(CAN_Type *base, uint8_t idx)

Gets the Tx buffer request pending status.

This function returns Tx Message Buffer transmission request pending status.

Parameters

- base – MCAN peripheral base address.

- idx – The MCAN Tx Buffer index.

uint32_t MCAN_IsTransmitOccurred(CAN_Type *base, uint8_t idx)

Gets the Tx buffer transmission occurred status.

This function returns Tx Message Buffer transmission occurred status.

Parameters

- base – MCAN peripheral base address.

- idx – The MCAN Tx Buffer index.

status_t MCAN_WriteTxBuffer(CAN_Type *base, uint8_t idx, const *mcan_tx_buffer_frame_t* *pTxFrame)

Writes an MCAN Message to the Transmit Buffer.

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

- base – MCAN peripheral base address.

- idx – The MCAN Tx Buffer index.

- pTxFrame – Pointer to CAN message frame to be sent.

status_t MCAN_ReadRxBuffer(CAN_Type *base, uint8_t idx, *mcan_rx_buffer_frame_t*
*pRxFrame)

Reads an MCAN Message from Rx Buffer.

This function reads a CAN message from the Rx Buffer in the Message RAM.

Parameters

- base – MCAN peripheral base address.

- idx – The MCAN Rx Buffer index.

- pRxFrame – Pointer to CAN message frame structure for reception.

Return values
kStatus_Success – - Read Message from Rx Buffer successfully.

status_t MCAN_ReadRxFifo(CAN_Type *base, uint8_t fifoBlock, *mcan_rx_buffer_frame_t*
*pRxFrame)

Reads an MCAN Message from Rx FIFO.

This function reads a CAN message from the Rx FIFO in the Message RAM.

Parameters

- base – MCAN peripheral base address.

- fifoBlock – Rx FIFO block 0 or 1.

- pRxFrame – Pointer to CAN message frame structure for reception.

Return values
kStatus_Success – - Read Message from Rx FIFO successfully.

static inline void MCAN_TransmitAddRequest(CAN_Type *base, uint8_t idx)

Tx Buffer add request to send message out.

This function add sending request to corresponding Tx Buffer.

Parameters

- base – MCAN peripheral base address.

- idx – Tx Buffer index.

static inline void MCAN_TransmitCancelRequest(CAN_Type *base, uint8_t idx)

Tx Buffer cancel sending request.

This function clears Tx buffer request pending bit.

Parameters

- base – MCAN peripheral base address.

- idx – Tx Buffer index.

status_t MCAN_TransferSendBlocking(CAN_Type *base, uint8_t idx, *mcan_tx_buffer_frame_t*
*pTxFrame)

Performs a polling send transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – MCAN peripheral base pointer.

- idx – The MCAN buffer index.

- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.

- kStatus_Fail – - Tx Message Buffer is currently in use.

status_t MCAN_TransferReceiveBlocking(CAN_Type *base, uint8_t idx, *mcan_rx_buffer_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – MCAN peripheral base pointer.

- idx – The MCAN buffer index.

- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Read Rx Message Buffer Successfully.

- kStatus_Fail – - No new message.

status_t MCAN_TransferReceiveFifoBlocking(CAN_Type *base, uint8_t fifoBlock, *mcan_rx_buffer_frame_t* *pRxFrame)

Performs a polling receive transaction from Rx FIFO on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – MCAN peripheral base pointer.

- fifoBlock – Rx FIFO block, 0 or 1.

- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.

- kStatus_Fail – - No new message in Rx FIFO.

void MCAN_TransferCreateHandle(CAN_Type *base, *mcan_handle_t* *handle, *mcan_transfer_callback_t* callback, void *userData)

Initializes the MCAN handle.

This function initializes the MCAN handle, which can be used for other MCAN transactional APIs. Usually, for a specified MCAN instance, call this API once to get the initialized handle.

Parameters

- base – MCAN peripheral base address.

- handle – MCAN handle pointer.

- callback – The callback function.

- userData – The parameter of the callback function.

status_t MCAN_TransferSendNonBlocking(CAN_Type *base, *mcan_handle_t* *handle,
                                                mcan_buffer_transfer_t *xfer)

> Sends a message using IRQ.

> This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

>> **Parameters**

>>> • base – MCAN peripheral base address.

>>> • handle – MCAN handle pointer.

>>> • xfer – MCAN Buffer transfer structure. See the mcan_buffer_transfer_t.

>> **Return values**

>>> • kStatus_Success – Start Tx Buffer sending process successfully.

>>> • kStatus_Fail – Write Tx Buffer failed.

>>> • kStatus_MCAN_TxBusy – Tx Buffer is in use.

status_t MCAN_TransferReceiveFifoNonBlocking(CAN_Type *base, uint8_t fifoBlock,
                                                mcan_handle_t *handle, *mcan_fifo_transfer_t*
                                                *xfer)

> Receives a message from Rx FIFO using IRQ.

> This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

>> **Parameters**

>>> • base – MCAN peripheral base address.

>>> • handle – MCAN handle pointer.

>>> • fifoBlock – Rx FIFO block, 0 or 1.

>>> • xfer – MCAN Rx FIFO transfer structure. See the mcan_fifo_transfer_t.

>> **Return values**

>>> • kStatus_Success – - Start Rx FIFO receiving process successfully.

>>> • kStatus_MCAN_RxFifo0Busy – - Rx FIFO 0 is currently in use.

>>> • kStatus_MCAN_RxFifo1Busy – - Rx FIFO 1 is currently in use.

void MCAN_TransferAbortSend(CAN_Type *base, *mcan_handle_t* *handle, uint8_t bufferIdx)

> Aborts the interrupt driven message send process.

> This function aborts the interrupt driven message send process.

>> **Parameters**

>>> • base – MCAN peripheral base address.

>>> • handle – MCAN handle pointer.

>>> • bufferIdx – The MCAN Buffer index.

void MCAN_TransferAbortReceiveFifo(CAN_Type *base, uint8_t fifoBlock, *mcan_handle_t*
                                                *handle)

> Aborts the interrupt driven message receive from Rx FIFO process.

> This function aborts the interrupt driven message receive from Rx FIFO process.

>> **Parameters**

>>> • base – MCAN peripheral base address.

> • fifoBlock – MCAN Fifo block, 0 or 1.
>
> • handle – MCAN handle pointer.

void MCAN_TransferHandleIRQ(CAN_Type *base, *mcan_handle_t* *handle)

MCAN IRQ handle function.

This function handles the MCAN Error, the Buffer, and the Rx FIFO IRQ request.

> **Parameters**
>
> > • base – MCAN peripheral base address.
> >
> > • handle – MCAN handle pointer.

FSL_MCAN_DRIVER_VERSION

MCAN driver version.

MCAN transfer status.

Values:

enumerator kStatus_MCAN_TxBusy

Tx Buffer is Busy.

enumerator kStatus_MCAN_TxIdle

Tx Buffer is Idle.

enumerator kStatus_MCAN_RxBusy

Rx Buffer is Busy.

enumerator kStatus_MCAN_RxIdle

Rx Buffer is Idle.

enumerator kStatus_MCAN_RxFifo0New

New message written to Rx FIFO 0.

enumerator kStatus_MCAN_RxFifo0Idle

Rx FIFO 0 is Idle.

enumerator kStatus_MCAN_RxFifo0Watermark

Rx FIFO 0 fill level reached watermark.

enumerator kStatus_MCAN_RxFifo0Full

Rx FIFO 0 full.

enumerator kStatus_MCAN_RxFifo0Lost

Rx FIFO 0 message lost.

enumerator kStatus_MCAN_RxFifo1New

New message written to Rx FIFO 1.

enumerator kStatus_MCAN_RxFifo1Idle

Rx FIFO 1 is Idle.

enumerator kStatus_MCAN_RxFifo1Watermark

Rx FIFO 1 fill level reached watermark.

enumerator kStatus_MCAN_RxFifo1Full

Rx FIFO 1 full.

enumerator kStatus_MCAN_RxFifo1Lost

Rx FIFO 1 message lost.

enumerator kStatus_MCAN_RxFifo0Busy
    Rx FIFO 0 is busy.

enumerator kStatus_MCAN_RxFifo1Busy
    Rx FIFO 1 is busy.

enumerator kStatus_MCAN_ErrorStatus
    MCAN Module Error and Status.

enumerator kStatus_MCAN_UnHandled
    UnHadled Interrupt asserted.

enum __mcan_flags
    MCAN status flags.

    This provides constants for the MCAN status flags for use in the MCAN functions. Note: The CPU read action clears MCAN_ErrorFlag, therefore user need to read MCAN_ErrorFlag and distinguish which error is occur using _mcan_error_flags enumerations.

    Values:

    enumerator kMCAN_AccesstoRsvdFlag
        CAN Synchronization Status.

    enumerator kMCAN_ProtocolErrDIntFlag
        Tx Warning Interrupt Flag.

    enumerator kMCAN_ProtocolErrAIntFlag
        Rx Warning Interrupt Flag.

    enumerator kMCAN_BusOffIntFlag
        Tx Error Warning Status.

    enumerator kMCAN_ErrorWarningIntFlag
        Rx Error Warning Status.

    enumerator kMCAN_ErrorPassiveIntFlag
        Rx Error Warning Status.

enum __mcan_rx_fifo_flags
    MCAN Rx FIFO status flags.

    The MCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO.

    Values:

    enumerator kMCAN_RxFifo0NewFlag
        Rx FIFO 0 new message flag.

    enumerator kMCAN_RxFifo0WatermarkFlag
        Rx FIFO 0 watermark reached flag.

    enumerator kMCAN_RxFifo0FullFlag
        Rx FIFO 0 full flag.

    enumerator kMCAN_RxFifo0LostFlag
        Rx FIFO 0 message lost flag.

    enumerator kMCAN_RxFifo1NewFlag
        Rx FIFO 0 new message flag.

    enumerator kMCAN_RxFifo1WatermarkFlag
        Rx FIFO 0 watermark reached flag.

enumerator kMCAN_RxFifo1FullFlag
    Rx FIFO 0 full flag.

enumerator kMCAN_RxFifo1LostFlag
    Rx FIFO 0 message lost flag.

enum __mcan_tx_flags
    MCAN Tx status flags.

    The MCAN Tx Status enumerations are used to determine the status of the Tx Buffer/Event
    FIFO.

    Values:

    enumerator kMCAN_TxTransmitCompleteFlag
        Transmission completed flag.

    enumerator kMCAN_TxTransmitCancelFinishFlag
        Transmission cancellation finished flag.

    enumerator kMCAN_TxEventFifoLostFlag
        Tx Event FIFO element lost.

    enumerator kMCAN_TxEventFifoFullFlag
        Tx Event FIFO full.

    enumerator kMCAN_TxEventFifoWatermarkFlag
        Tx Event FIFO fill level reached watermark.

    enumerator kMCAN_TxEventFifoNewFlag
        Tx Handler wrote Tx Event FIFO element flag.

    enumerator kMCAN_TxEventFifoEmptyFlag
        Tx FIFO empty flag.

enum __mcan_interrupt_enable
    MCAN interrupt configuration structure, default settings all disabled.

    This structure contains the settings for all of the MCAN Module interrupt configurations.

    Values:

    enumerator kMCAN_BusOffInterruptEnable
        Bus Off interrupt.

    enumerator kMCAN_ErrorInterruptEnable
        Error interrupt.

    enumerator kMCAN_WarningInterruptEnable
        Rx Warning interrupt.

enum __mcan_frame_idformat
    MCAN frame format.

    Values:

    enumerator kMCAN_FrameIDStandard
        Standard frame format attribute.

    enumerator kMCAN_FrameIDExtend
        Extend frame format attribute.

enum __mcan_frame_type
    MCAN frame type.

    Values:

enumerator kMCAN_FrameTypeData
    Data frame type attribute.

enumerator kMCAN_FrameTypeRemote
    Remote frame type attribute.

enum __mcan_bytes_in_datafield
    MCAN frame datafield size.

    Values:

    enumerator kMCAN_8ByteDatafield
        8 byte data field.

    enumerator kMCAN_12ByteDatafield
        12 byte data field.

    enumerator kMCAN_16ByteDatafield
        16 byte data field.

    enumerator kMCAN_20ByteDatafield
        20 byte data field.

    enumerator kMCAN_24ByteDatafield
        24 byte data field.

    enumerator kMCAN_32ByteDatafield
        32 byte data field.

    enumerator kMCAN_48ByteDatafield
        48 byte data field.

    enumerator kMCAN_64ByteDatafield
        64 byte data field.

enum __mcan_fifo_type
    MCAN Rx FIFO block number.

    Values:

    enumerator kMCAN_Fifo0
        CAN Rx FIFO 0.

    enumerator kMCAN_Fifo1
        CAN Rx FIFO 1.

enum __mcan_fifo_opmode_config
    MCAN FIFO Operation Mode.

    Values:

    enumerator kMCAN_FifoBlocking
        FIFO blocking mode.

    enumerator kMCAN_FifoOverwrite
        FIFO overwrite mode.

enum __mcan_txmode_config
    MCAN Tx FIFO/Queue Mode.

    Values:

    enumerator kMCAN_txFifo
        Tx FIFO operation.

enumerator kMCAN_txQueue
> Tx Queue operation.

enum __mcan_remote_frame_config
> MCAN remote frames treatment.

> *Values:*

enumerator kMCAN_filterFrame
> Filter remote frames.

enumerator kMCAN_rejectFrame
> Reject all remote frames.

enum __mcan_nonmasking_frame_config
> MCAN non-masking frames treatment.

> *Values:*

enumerator kMCAN_acceptinFifo0
> Accept non-masking frames in Rx FIFO 0.

enumerator kMCAN_acceptinFifo1
> Accept non-masking frames in Rx FIFO 1.

enumerator kMCAN_reject0
> Reject non-masking frames.

enumerator kMCAN_reject1
> Reject non-masking frames.

enum __mcan_fec_config
> MCAN Filter Element Configuration.

> *Values:*

enumerator kMCAN_disable
> Disable filter element.

enumerator kMCAN_storeinFifo0
> Store in Rx FIFO 0 if filter matches.

enumerator kMCAN_storeinFifo1
> Store in Rx FIFO 1 if filter matches.

enumerator kMCAN_reject
> Reject ID if filter matches.

enumerator kMCAN_setprio
> Set priority if filter matches.

enumerator kMCAN_setpriofifo0
> Set priority and store in FIFO 0 if filter matches.

enumerator kMCAN_setpriofifo1
> Set priority and store in FIFO 1 if filter matches.

enumerator kMCAN_storeinbuffer
> Store into Rx Buffer or as debug message.

enum __mcan_std_filter_type
> MCAN Filter Type.

> *Values:*

enumerator kMCAN_range
Range filter from SFID1 to SFID2.

enumerator kMCAN_dual
Dual ID filter for SFID1 or SFID2.

enumerator kMCAN_classic
Classic filter: SFID1 = filter, SFID2 = mask.

enumerator kMCAN_disableORrange2
Filter element disabled for standard filter or Range filter, XIDAM mask not applied for extended filter.

typedef enum _mcan_frame_idformat mcan_frame_idformat_t
MCAN frame format.

typedef enum _mcan_frame_type mcan_frame_type_t
MCAN frame type.

typedef enum _mcan_bytes_in_datafield mcan_bytes_in_datafield_t
MCAN frame datafield size.

typedef struct _mcan_tx_buffer_frame mcan_tx_buffer_frame_t
MCAN Tx Buffer structure.

typedef struct _mcan_rx_buffer_frame mcan_rx_buffer_frame_t
MCAN Rx FIFO/Buffer structure.

typedef enum _mcan_fifo_type mcan_fifo_type_t
MCAN Rx FIFO block number.

typedef enum _mcan_fifo_opmode_config mcan_fifo_opmode_config_t
MCAN FIFO Operation Mode.

typedef enum _mcan_txmode_config mcan_txmode_config_t
MCAN Tx FIFO/Queue Mode.

typedef enum _mcan_remote_frame_config mcan_remote_frame_config_t
MCAN remote frames treatment.

typedef enum _mcan_nonmasking_frame_config mcan_nonmasking_frame_config_t
MCAN non-masking frames treatment.

typedef enum _mcan_fec_config mcan_fec_config_t
MCAN Filter Element Configuration.

typedef struct _mcan_rx_fifo_config mcan_rx_fifo_config_t
MCAN Rx FIFO configuration.

typedef struct _mcan_rx_buffer_config mcan_rx_buffer_config_t
MCAN Rx Buffer configuration.

typedef struct _mcan_tx_fifo_config mcan_tx_fifo_config_t
MCAN Tx Event FIFO configuration.

typedef struct _mcan_tx_buffer_config mcan_tx_buffer_config_t
MCAN Tx Buffer configuration.

typedef enum _mcan_std_filter_type mcan_filter_type_t
MCAN Filter Type.

typedef struct _mcan_std_filter_element_config mcan_std_filter_element_config_t
MCAN Standard Message ID Filter Element.

typedef struct _mcan_ext_filter_element_config mcan_ext_filter_element_config_t
    MCAN Extended Message ID Filter Element.

typedef struct _mcan_frame_filter_config mcan_frame_filter_config_t
    MCAN Rx filter configuration.

typedef struct _mcan_timing_config mcan_timing_config_t
    MCAN protocol timing characteristic configuration structure.

typedef struct _mcan_timing_param mcan_timing_param_t
    MCAN bit timing parameter configuration structure.

typedef struct _mcan_memory_config mcan_memory_config_t
    MCAN Message RAM related configuration structure.

typedef struct _mcan_config mcan_config_t
    MCAN module configuration structure.

typedef struct _mcan_buffer_transfer mcan_buffer_transfer_t
    MCAN Buffer transfer.

typedef struct _mcan_fifo_transfer mcan_fifo_transfer_t
    MCAN Rx FIFO transfer.

typedef struct _mcan_handle mcan_handle_t
    MCAN handle structure definition.

typedef void (*mcan_transfer_callback_t)(CAN_Type *base, *mcan_handle_t* *handle, *status_t* status, uint32_t result, void *userData)
    MCAN transfer callback function.

    The MCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_MCAN_ErrorStatus, the result parameter is the Content of MCAN status register which can be used to get the working status(or error status) of MCAN module. If the status equals to other MCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other MCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

MCAN_RETRY_TIMES

struct __mcan_tx_buffer_frame
    #include <fsl_mcan.h> MCAN Tx Buffer structure.

Public Members

uint8_t size
    classical CAN is 8(bytes), FD is 12/64 such.

struct __mcan_rx_buffer_frame
    #include <fsl_mcan.h> MCAN Rx FIFO/Buffer structure.

Public Members

uint8_t size
    classical CAN is 8(bytes), FD is 12/64 such.

struct __mcan_rx_fifo_config
    #include <fsl_mcan.h> MCAN Rx FIFO configuration.

Public Members

uint32_t address
    FIFOn start address.

uint32_t elementSize
    FIFOn element number.

uint32_t watermark
    FIFOn watermark level.

mcan_fifo_opmode_config_t opmode
    FIFOn blocking/overwrite mode.

mcan_bytes_in_datafield_t datafieldSize
    Data field size per frame, size>8 is for CANFD.

struct __mcan__rx__buffer__config
    #include <fsl_mcan.h> MCAN Rx Buffer configuration.

Public Members

uint32_t address
    Rx Buffer start address.

mcan_bytes_in_datafield_t datafieldSize
    Data field size per frame, size>8 is for CANFD.

struct __mcan__tx__fifo__config
    #include <fsl_mcan.h> MCAN Tx Event FIFO configuration.

Public Members

uint32_t address
    Event fifo start address.

uint32_t elementSize
    FIFOn element number.

uint32_t watermark
    FIFOn watermark level.

struct __mcan__tx__buffer__config
    #include <fsl_mcan.h> MCAN Tx Buffer configuration.

Public Members

uint32_t address
    Tx Buffers Start Address.

uint32_t dedicatedSize
    Number of Dedicated Transmit Buffers.

uint32_t fqSize
    Transmit FIFO/Queue Size.

mcan_txmode_config_t mode
    Tx FIFO/Queue Mode.

        mcan_bytes_in_datafield_t datafieldSize

            Data field size per frame, size>8 is for CANFD.

struct __mcan__std__filter__element__config

        #include <fsl_mcan.h> MCAN Standard Message ID Filter Element.

Public Members

uint32_t sfid2

        Standard Filter ID 2.

uint32_t ___pad0___

        Reserved.

uint32_t sfid1

        Standard Filter ID 1.

uint32_t sfec

        Standard Filter Element Configuration.

uint32_t sft

        Standard Filter Type.

struct __mcan__ext__filter__element__config

        #include <fsl_mcan.h> MCAN Extended Message ID Filter Element.

Public Members

uint32_t efid1

        Extended Filter ID 1.

uint32_t efec

        Extended Filter Element Configuration.

uint32_t efid2

        Extended Filter ID 2.

uint32_t ___pad0___

        Reserved.

uint32_t eft

        Extended Filter Type.

struct __mcan__frame__filter__config

        #include <fsl_mcan.h> MCAN Rx filter configuration.

Public Members

uint32_t address

        Filter start address.

uint32_t listSize

        Filter list size.

mcan_frame_idformat_t idFormat

        Frame format.

mcan_remote_frame_config_t remFrame

        Remote frame treatment.

mcan_nonmasking_frame_config_t nmFrame
    Non-masking frame treatment.

struct __mcan__timing__config
    #include <fsl_mcan.h> MCAN protocol timing characteristic configuration structure.

Public Members

uint16_t preDivider
    Nominal Clock Pre-scaler Division Factor.

uint8_t rJumpwidth
    Nominal Re-sync Jump Width.

uint8_t seg1
    Nominal Time Segment 1.

uint8_t seg2
    Nominal Time Segment 2.

uint16_t datapreDivider
    Data Clock Pre-scaler Division Factor.

uint8_t datarJumpwidth
    Data Re-sync Jump Width.

uint8_t dataseg1
    Data Time Segment 1.

uint8_t dataseg2
    Data Time Segment 2.

struct __mcan__timing__param
    #include <fsl_mcan.h> MCAN bit timing parameter configuration structure.

Public Members

uint32_t busLength
    Maximum Bus length in meter.

uint32_t propTxRx
    Transceiver propagation delay in nanosecond.

uint32_t nominalbaudRate
    Baud rate of Arbitration phase in bps.

uint32_t nominalSP
    Sample point of Arbitration phase, range in 10 ~ 990, 800 means 80%.

uint32_t databaudRate
    Baud rate of Data phase in bps.

uint32_t dataSP
    Sample point of Data phase, range in 0 ~ 1000, 800 means 80%.

struct __mcan__memory__config
    #include <fsl_mcan.h> MCAN Message RAM related configuration structure.

Public Members

uint32_t baseAddr
    Message RAM base address, should be 4k alignment.

struct __mcan__config
    #include <fsl_mcan.h> MCAN module configuration structure.

Public Members

uint32_t baudRateA
    Baud rate of Arbitration phase in bps.

uint32_t baudRateD
    Baud rate of Data phase in bps.

bool enableCanfdNormal
    Enable or Disable CANFD normal.

bool enableCanfdSwitch
    Enable or Disable CANFD with baudrate switch.

bool enableLoopBackInt
    Enable or Disable Internal Back.

bool enableLoopBackExt
    Enable or Disable External Loop Back.

bool enableBusMon
    Enable or Disable Bus Monitoring Mode.

mcan_timing_config_t timingConfig
    Protocol timing .

struct __mcan__buffer__transfer
    #include <fsl_mcan.h> MCAN Buffer transfer.

Public Members

mcan_tx_buffer_frame_t *frame
    The buffer of CAN Message to be transfer.

uint8_t bufferIdx
    The index of Message buffer used to transfer Message.

struct __mcan__fifo__transfer
    #include <fsl_mcan.h> MCAN Rx FIFO transfer.

Public Members

mcan_rx_buffer_frame_t *frame
    The buffer of CAN Message to be received from Rx FIFO.

struct __mcan__handle
    #include <fsl_mcan.h> MCAN handle structure.

Public Members

mcan_transfer_callback_t callback
   Callback function.

void *userData
   MCAN callback function parameter.

mcan_tx_buffer_frame_t *volatile bufferFrameBuf[64]
   The buffer for received data from Buffers.

mcan_rx_buffer_frame_t *volatile rxFifoFrameBuf
   The buffer for received data from Rx FIFO.

volatile uint8_t bufferState[64]
   Message Buffer transfer state.

volatile uint8_t rxFifoState
   Rx FIFO transfer state.

struct ___unnamed11___

Public Members

uint32_t id
   CAN Frame Identifier.

uint32_t rtr
   CAN Frame Type(DATA or REMOTE).

uint32_t xtd
   CAN Frame Type(STD or EXT).

uint32_t esi
   CAN Frame Error State Indicator.

struct ___unnamed13___

Public Members

uint32_t dlc
   Data Length Code 9 10 11 12 13 14 15 Number of data bytes 12 16 20 24 32 48 64

uint32_t brs
   Bit Rate Switch.

uint32_t fdf
   CAN FD format.

uint32_t ___pad1___
   Reserved.

uint32_t efc
   Event FIFO control.

uint32_t mm
   Message Marker.

struct ___unnamed15___

Public Members

uint32_t id
    CAN Frame Identifier.

uint32_t rtr
    CAN Frame Type(DATA or REMOTE).

uint32_t xtd
    CAN Frame Type(STD or EXT).

uint32_t esi
    CAN Frame Error State Indicator.

struct ___unnamed17___

Public Members

uint32_t rxts
    Rx Timestamp.

uint32_t dlc
    Data Length Code 9 10 11 12 13 14 15 Number of data bytes 12 16 20 24 32 48 64

uint32_t brs
    Bit Rate Switch.

uint32_t fdf
    CAN FD format.

uint32_t ___pad0___
    Reserved.

uint32_t fidx
    Filter Index.

uint32_t anmf
    Accepted Non-matching Frame.

2.31 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 reigster, 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

2.32   OTP: One-Time Programmable memory and API

FSL_OTP_DRIVER_VERSION
OTP driver version 2.0.1.

Current version: 2.0.1

Change log:

- Version 2.0.1
  - Fixed MISRA-C 2012 violations.
- Version 2.0.0
  - Initial version.

enum __otp_bank
Bank bit flags.

Values:

enumerator kOTP_Bank0
Bank 0.

enumerator kOTP_Bank1
Bank 1.

enumerator kOTP_Bank2
Bank 2.

enumerator kOTP_Bank3
Bank 3.

enum __otp_word
Bank word bit flags.

Values:

enumerator kOTP_Word0
Word 0.

enumerator kOTP_Word1
Word 1.

enumerator kOTP_Word2
Word 2.

enumerator kOTP_Word3
Word 3.

enum __otp_lock
Lock modifications of a read or write access to a bank register.

Values:

enumerator kOTP_LockDontLock
Do not lock.

enumerator kOTP_LockLock
Lock till reset.

enum __otp_status

OTP error codes.

Values:

enumerator kStatus_OTP_WrEnableInvalid
Write enable invalid.

enumerator kStatus_OTP_SomeBitsAlreadyProgrammed
Some bits already programmed.

enumerator kStatus_OTP_AllDataOrMaskZero
All data or mask zero.

enumerator kStatus_OTP_WriteAccessLocked
Write access locked.

enumerator kStatus_OTP_ReadDataMismatch
Read data mismatch.

enumerator kStatus_OTP_UsbIdEnabled
USB ID enabled.

enumerator kStatus_OTP_EthMacEnabled
Ethernet MAC enabled.

enumerator kStatus_OTP_AesKeysEnabled
AES keys enabled.

enumerator kStatus_OTP_IllegalBank
Illegal bank.

enumerator kStatus_OTP_ShufflerConfigNotValid
Shuffler config not valid.

enumerator kStatus_OTP_ShufflerNotEnabled
Shuffler not enabled.

enumerator kStatus_OTP_ShufflerCanOnlyProgSingleKey
Shuffler can only program single key.

enumerator kStatus_OTP_IllegalProgramData
Illegal program data.

enumerator kStatus_OTP_ReadAccessLocked
Read access locked.

typedef enum *_otp_bank* otp_bank_t
Bank bit flags.

typedef enum *_otp_word* otp_word_t
Bank word bit flags.

typedef enum *_otp_lock* otp_lock_t
Lock modifications of a read or write access to a bank register.

static inline *status_t* OTP_Init(void)
Initializes OTP controller.

Returns
kStatus_Success upon successful execution, error status otherwise.

static inline *status_t* OTP_EnableBankWriteMask(*otp_bank_t* bankMask)

Unlock one or more OTP banks for write access.

Parameters

- bankMask – bit flag that specifies which banks to unlock.

Returns

kStatus_Success upon successful execution, error status otherwise.

static inline *status_t* OTP_DisableBankWriteMask(*otp_bank_t* bankMask)

Lock one or more OTP banks for write access.

Parameters

- bankMask – bit flag that specifies which banks to lock.

Returns

kStatus_Success upon successful execution, error status otherwise.

static inline *status_t* OTP_EnableBankWriteLock(uint32_t bankIndex, *otp_word_t* regEnableMask, *otp_word_t* regDisableMask, *otp_lock_t* lockWrite)

Locks or unlocks write access to a register of an OTP bank and possibly lock un/locking of it.

Parameters

- bankIndex – OTP bank index, 0 = bank 0, 1 = bank 1 etc.

- regEnableMask – bit flag that specifies for which words to enable writing.

- regDisableMask – bit flag that specifies for which words to disable writing.

- lockWrite – specifies if access set can be modified or is locked till reset.

Returns

kStatus_Success upon successful execution, error status otherwise.

static inline *status_t* OTP_EnableBankReadLock(uint32_t bankIndex, *otp_word_t* regEnableMask, *otp_word_t* regDisableMask, *otp_lock_t* lockWrite)

Locks or unlocks read access to a register of an OTP bank and possibly lock un/locking of it.

Parameters

- bankIndex – OTP bank index, 0 = bank 0, 1 = bank 1 etc.

- regEnableMask – bit flag that specifies for which words to enable reading.

- regDisableMask – bit flag that specifies for which words to disable reading.

- lockWrite – specifies if access set can be modified or is locked till reset.

Returns

kStatus_Success upon successful execution, error status otherwise.

static inline *status_t* OTP_ProgramRegister(uint32_t bankIndex, uint32_t regIndex, uint32_t value)

Program a single register in an OTP bank.

Parameters

- bankIndex – OTP bank index, 0 = bank 0, 1 = bank 1 etc.

- regIndex – OTP register index.

- value – value to write.

Returns
kStatus_Success upon successful execution, error status otherwise.

static inline uint32_t OTP_GetDriverVersion(**void**)

Returns the version of the OTP driver in ROM.

Returns
version.

FSL_COMPONENT_ID

_OTP_ERR_BASE

_OTP_MAKE_STATUS(**errorCode**)

2.33   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.

2.33. PINT: Pin Interrupt and Pattern Match Driver 355

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 interupt 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.

2.33. PINT: Pin Interrupt and Pattern Match Driver

> **Return values**
>> None. –

static inline uint32_t PINT_PinInterruptGetFallFlag(PINT_Type *base, *pint_pin_int_t* pintr)

> Get selected pin interrupt fall flag.
>
> This function returns the selected pin interrupt fall flag.
>
>> **Parameters**
>>
>>> - base – Base address of the PINT peripheral.
>>>
>>> - pintr – Pin interrupt.
>>
>> **Return values**
>>> flag – = 0 Falling edge has not been detected. = 1 Falling edge has been detected.

static inline void PINT_PinInterruptClrFallFlagAll(PINT_Type *base)

> Clear all pin interrupt fall flags.
>
> This function clears the fall flag for all pin interrupts.
>
>> **Parameters**
>>
>>> - base – Base address of the PINT peripheral.
>>
>> **Return values**
>>> None. –

static inline uint32_t PINT_PinInterruptGetFallFlagAll(PINT_Type *base)

> Get all pin interrupt fall flags.
>
> This function returns the fall flag of all pin interrupts.
>
>> **Parameters**
>>
>>> - base – Base address of the PINT peripheral.
>>
>> **Return values**
>>> flags – Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

static inline void PINT_PinInterruptClrRiseFlag(PINT_Type *base, *pint_pin_int_t* pintr)

> Clear Selected pin interrupt rise flag.
>
> This function clears the selected pin interrupt rise flag.
>
>> **Parameters**
>>
>>> - base – Base address of the PINT peripheral.
>>>
>>> - pintr – Pin interrupt.
>>
>> **Return values**
>>> None. –

static inline uint32_t PINT_PinInterruptGetRiseFlag(PINT_Type *base, *pint_pin_int_t* pintr)

> Get selected pin interrupt rise flag.
>
> This function returns the selected pin interrupt rise flag.
>
>> **Parameters**
>>
>>> - base – Base address of the PINT peripheral.
>>>
>>> - pintr – Pin interrupt.
>>
>> **Return values**
>>> flag – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

static inline void PINT_PinInterruptClrRiseFlagAll(PINT_Type *base)

> Clear all pin interrupt rise flags.

> This function clears the rise flag for all pin interrupts.

>> **Parameters**
>>> • base – Base address of the PINT peripheral.

>> **Return values**
>>> None. –

static inline uint32_t PINT_PinInterruptGetRiseFlagAll(PINT_Type *base)

> Get all pin interrupt rise flags.

> This function returns the rise flag of all pin interrupts.

>> **Parameters**
>>> • base – Base address of the PINT peripheral.

>> **Return values**
>>> flags – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

void PINT_PatternMatchConfig(PINT_Type *base, *pint_pmatch_bslice_t* bslice, *pint_pmatch_cfg_t* *cfg)

> Configure PINT pattern match.

> This function configures a given pattern match bit slice.

>> **Parameters**
>>> • base – Base address of the PINT peripheral.

>>> • bslice – Pattern match bit slice number.

>>> • cfg – Pointer to bit slice configuration.

>> **Return values**
>>> None. –

void PINT_PatternMatchGetConfig(PINT_Type *base, *pint_pmatch_bslice_t* bslice, *pint_pmatch_cfg_t* *cfg)

> Get PINT pattern match configuration.

> This function returns the configuration of a given pattern match bit slice.

>> **Parameters**
>>> • base – Base address of the PINT peripheral.

>>> • bslice – Pattern match bit slice number.

>>> • cfg – Pointer to bit slice configuration.

>> **Return values**
>>> None. –

static inline uint32_t PINT_PatternMatchGetStatus(PINT_Type *base, *pint_pmatch_bslice_t* bslice)

> Get pattern match bit slice status.

> This function returns the status of selected bit slice.

>> **Parameters**
>>> • base – Base address of the PINT peripheral.

>>> • bslice – Pattern match bit slice number.

2.33. PINT: Pin Interrupt and Pattern Match Driver 359

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_EnableInterruptByIndex(PINT_Type *base, *pint_pin_int_t* pintIdx)

> enable interrupt in NVIC by pin index.

> This function enables the interrupt in the NVIC. The difference with PINT_EnableCallbackByIndex() is that PINT_EnableCallbackByIndex() not only enables the interrupt in the NVIC but also clears pending interrupts. Use this function together with PINT_DisableInterruptByIndex() to temporarily disable/enable the pin interrupt. Use PINT_EnableCallbackByIndex() to enable the interrupt after installing the callback.

> > **Parameters**

> > > • base – Base address of the peripheral.

- pinIdx – pin index.

Return values

None. –

void PINT_DisableInterruptByIndex(PINT_Type *base, *pint_pin_int_t* pintIdx)

disable interrupt in NVIC by pin index.

This function disables the interrupt in the NVIC. The difference with PINT_DisableCallbackByIndex() is that PINT_DisableCallbackByIndex() not only disables the interrupt in the NVIC but also clears pending interrupts. Use this function together with PINT_EnableInterruptByIndex() to temporarily disable/enable the pin interrupt. Use PINT_DisableCallbackByIndex() to disable the interrupt in a de-init function.

Parameters

- base – Base address of the peripheral.
- pinIdx – 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>

2.34 Power Driver

enum pd_bits
    Values:

    enumerator kPDRUNCFG_LP_REG

    enumerator kPDRUNCFG_PD_FRO_EN

    enumerator kPDRUNCFG_PD_TS

    enumerator kPDRUNCFG_PD_BOD_RESET

    enumerator kPDRUNCFG_PD_BOD_INTR

    enumerator kPDRUNCFG_PD_VD2_ANA

    enumerator kPDRUNCFG_PD_ADC0

    enumerator kPDRUNCFG_PD_RAM0

    enumerator kPDRUNCFG_PD_RAM1

    enumerator kPDRUNCFG_PD_RAM2

    enumerator kPDRUNCFG_PD_RAM3

    enumerator kPDRUNCFG_PD_ROM

    enumerator kPDRUNCFG_PD_VDDA

    enumerator kPDRUNCFG_PD_WDT_OSC

    enumerator kPDRUNCFG_PD_USB0_PHY

    enumerator kPDRUNCFG_PD_SYS_PLL0

    enumerator kPDRUNCFG_PD_VREFP

    enumerator kPDRUNCFG_PD_FLASH_BG

    enumerator kPDRUNCFG_PD_VD3

    enumerator kPDRUNCFG_PD_VD4

    enumerator kPDRUNCFG_PD_VD5

    enumerator kPDRUNCFG_PD_VD6

    enumerator kPDRUNCFG_REQ_DELAY

    enumerator kPDRUNCFG_FORCE_RBB

    enumerator kPDRUNCFG_PD_USB1_PHY

    enumerator kPDRUNCFG_PD_USB_PLL

enumerator kPDRUNCFG_PD_AUDIO_PLL

enumerator kPDRUNCFG_PD_SYS_OSC

enumerator kPDRUNCFG_PD_EEPROM

enumerator kPDRUNCFG_PD_rng

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: 1.62V.

enumerator kBod_ResetLevel1
Reset Level0: 1.68V.

enumerator kBod_ResetLevel2
Reset Level0: 2.21V.

enumerator kBod_ResetLevel3
Reset Level0: 2.85V.

enum __power_bod_interrupt_level
The enumeration of BOD interrupt level.

Values:

enumerator kBod_InterruptLevel0
Interrupt level: 1.63V.

enumerator kBod_InterruptLevel1
Interrupt level: 1.68V.

enumerator kBod_InterruptLevel2
Interrupt level: 1.95V.

enumerator kBod_InterruptLevel3
Interrupt level: 2.86V.

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 2.2.0.

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

void POWER_OtpReload(**void**)
> Power Library API to reload OTP. This API must be called if VD6 is power down and power back again since FROHF TRIM value is store in OTP. If not, when calling FROHF settng API in clock driver then the FROHF clock out put will be inaccurate.

> > **Returns**
> > > none

void POWER__SetPLL(void)

> Power Library API to power the PLLs.

> > **Returns**
> > none

void POWER__SetUsbPhy(void)

> Power Library API to power the USB PHY.

> > **Returns**
> > none

void POWER__EnterPowerMode(*power_mode_cfg_t* mode, uint64_t exclude_from_pd)

> Power Library API to enter different power modes.

> > **Parameters**

> > > • mode – power mode

> > > • exclude_from_pd – Bit mask of the PDRUNCFG0(low 32bits) and PDRUNCFG1(high 32bits) that needs to be powered on during power mode selected.

> > **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 PDRUNCFG0(low 32bits) and PDRUNCFG1(high 32bits) 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 PDRUNCFG0(low 32bits) and PDRUNCFG1(high 32bits) that needs to be powered on during deep power down mode, but this is 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

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 the 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 the 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

uint32_t

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.

2.35 PUF: Physical Unclonable Function

FSL_PUF_DRIVER_VERSION

> PUF driver version. Version 2.2.0.
>
> Current version: 2.2.0
>
> Change log:
>
> - 2.0.0
>   - Initial version.
> - 2.0.1
>   - Fixed puf_wait_usec function optimization issue.
> - 2.0.2
>   - Add PUF configuration structure and support for PUF SRAM controller. Remove magic constants.
> - 2.0.3
>   - Fix MISRA C-2012 issue.
> - 2.1.0
>   - Align driver with PUF SRAM controller registers on LPCXpresso55s16.
>   - Update initizalition logic .
> - 2.1.1
>   - Fix ARMGCC build warning .
> - 2.1.2
>   - Update: Add automatic big to little endian swap for user (pre-shared) keys destinated to secret hardware bus (PUF key index 0).
> - 2.1.3
>   - Fix MISRA C-2012 issue.
> - 2.1.4
>   - Replace register uint32_t ticksCount with volatile uint32_t ticksCount in puf_wait_usec() to prevent optimization out delay loop.
> - 2.1.5
>   - Use common SDK delay in puf_wait_usec()
> - 2.1.6
>   - Changed wait time in PUF_Init(), when initialization fails it will try PUF_Powercycle() with shorter time. If this shorter time will also fail, initialization will be tried with worst case time as before.
> - 2.2.0
> - Add support for kPUF_KeySlot4.
> - Add new PUF_ClearKey() function, that clears a desired PUF internal HW key register.

enum __puf_key_index_register

> *Values:*
>
> enumerator kPUF_KeyIndex_00
>
> enumerator kPUF_KeyIndex_01
>
> enumerator kPUF_KeyIndex_02

enumerator kPUF_KeyIndex_03

enumerator kPUF_KeyIndex_04

enumerator kPUF_KeyIndex_05

enumerator kPUF_KeyIndex_06

enumerator kPUF_KeyIndex_07

enumerator kPUF_KeyIndex_08

enumerator kPUF_KeyIndex_09

enumerator kPUF_KeyIndex_10

enumerator kPUF_KeyIndex_11

enumerator kPUF_KeyIndex_12

enumerator kPUF_KeyIndex_13

enumerator kPUF_KeyIndex_14

enumerator kPUF_KeyIndex_15

enum _puf_min_max
> *Values:*

enumerator kPUF_KeySizeMin

enumerator kPUF_KeySizeMax

enumerator kPUF_KeyIndexMax

enum _puf_key_slot
> PUF key slot.

> *Values:*

enumerator kPUF_KeySlot0
> PUF key slot 0

enumerator kPUF_KeySlot1
> PUF key slot 1

PUF status return codes.
> *Values:*

enumerator kStatus_EnrollNotAllowed

enumerator kStatus_StartNotAllowed

typedef enum *_puf_key_index_register* puf_key_index_register_t

typedef enum *_puf_min_max* puf_min_max_t

typedef enum *_puf_key_slot* puf_key_slot_t
> PUF key slot.

PUF_GET_KEY_CODE_SIZE_FOR_KEY_SIZE(x)
> Get Key Code size in bytes from key size in bytes at compile time.

PUF_MIN_KEY_CODE_SIZE

PUF_ACTIVATION_CODE_SIZE

KEYSTORE_PUF_DISCHARGE_TIME_FIRST_TRY_MS

KEYSTORE_PUF_DISCHARGE_TIME_MAX_MS

struct puf_config_t
> *#include <fsl_puf.h>*

2.36 Reset Driver

enum __SYSCON_RSTn
> Enumeration for peripheral reset control bits.
>
> Defines the enumeration for peripheral reset control bits in PRESETC-TRL/ASYNCPRESETCTRL registers
>
> *Values:*
>
> enumerator kSPIFI_RST_SHIFT_RSTn
> > SPIFI 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 kGPIO2_RST_SHIFT_RSTn
> > GPIO2 reset control
>
> enumerator kGPIO3_RST_SHIFT_RSTn
> > GPIO3 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 kMCAN0_RST_SHIFT_RSTn
    MCAN0 reset control

enumerator kMCAN1_RST_SHIFT_RSTn
    MCAN1 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 kDMIC_RST_SHIFT_RSTn
    Digital microphone interface reset control

enumerator kCT32B2_RST_SHIFT_RSTn
    CT32B2 reset control

enumerator kUSB0D_RST_SHIFT_RSTn
    USB0D reset control

enumerator kCT32B0_RST_SHIFT_RSTn
    CT32B0 reset control

enumerator kCT32B1_RST_SHIFT_RSTn
    CT32B1 reset control

enumerator kLCD_RST_SHIFT_RSTn
    LCD reset control

enumerator kSDIO_RST_SHIFT_RSTn
    SDIO reset control

enumerator kUSB1H_RST_SHIFT_RSTn
    USB1H reset control

enumerator kUSB1D_RST_SHIFT_RSTn
    USB1D reset control

enumerator kUSB1RAM_RST_SHIFT_RSTn
    USB1RAM reset control

enumerator kEMC_RST_SHIFT_RSTn
    EMC reset control

enumerator kETH_RST_SHIFT_RSTn
    ETH reset control

enumerator kGPIO4_RST_SHIFT_RSTn
    GPIO4 reset control

enumerator kGPIO5_RST_SHIFT_RSTn
    GPIO5 reset control

enumerator kAES_RST_SHIFT_RSTn
    AES reset control

enumerator kOTP_RST_SHIFT_RSTn
    OTP reset control

enumerator kRNG_RST_SHIFT_RSTn
    RNG reset control

enumerator kFC8_RST_SHIFT_RSTn
    Flexcomm Interface 8 reset control

enumerator kFC9_RST_SHIFT_RSTn
    Flexcomm Interface 9 reset control

enumerator kUSB0HMR_RST_SHIFT_RSTn
    USB0HMR reset control

enumerator kUSB0HSL_RST_SHIFT_RSTn
    USB0HSL reset control

enumerator kSHA_RST_SHIFT_RSTn
    SHA reset control

enumerator kSC0_RST_SHIFT_RSTn
    SC0 reset control

enumerator kSC1_RST_SHIFT_RSTn
    SC1 reset control

enumerator kFC10_RST_SHIFT_RSTn
    Flexcomm Interface 10 reset control

enumerator kPUF_RST_SHIFT_RSTn
    PUF reset control

enumerator kCT32B3_RST_SHIFT_RSTn
    CT32B3 reset control

enumerator kCT32B4_RST_SHIFT_RSTn
    CT32B4 reset control

typedef enum *_SYSCON_RSTn* SYSCON_RSTn_t

    Enumeration for peripheral reset control bits.

    Defines the enumeration for peripheral reset control bits in PRESETC-TRL/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

AES_RSTS

CRC_RSTS

CTIMER_RSTS

DMA_RSTS_N

DMIC_RSTS

EMC_RSTS

ETH_RST

FLEXCOMM_RSTS

GINT_RSTS

GPIO_RSTS_N

INPUTMUX_RSTS

IOCON_RSTS

FLASH_RSTS

LCD_RSTS

MRT_RSTS

MCAN_RSTS

OTP_RSTS

PINT_RSTS

RNG_RSTS

SDIO_RST

SCT_RSTS

SHA_RST

SPIFI_RSTS

USB0D_RST

USB0HMR_RST

USB0HSL_RST

USB1H_RST

USB1D_RST

USB1RAM_RST

UTICK_RSTS

WWDT_RSTS

USB1RAM_RSTS

USB1H_RSTS

USB1D_RSTS

USB0HSL_RSTS

USB0HMR_RSTS

USB0D_RSTS

SHA_RSTS

SDIO_RSTS

ETH_RSTS

2.37   RIT: Repetitive Interrupt Timer

void RIT_Init(RIT_Type *base, const *rit_config_t* *config)

> Ungates the RIT clock, enables the RIT module, and configures the peripheral for basic operations.

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

Parameters

- base – RIT peripheral base address
- config – Pointer to the user's RIT config structure

void RIT_Deinit(RIT_Type *base)

> Gates the RIT clock and disables the RIT module.

Parameters

- base – RIT peripheral base address

void RIT_GetDefaultConfig(*rit_config_t* *config)

> Fills in the RIT configuration structure with the default settings.
>
> The default values are as follows.

```
config->enableRunInDebug = false;
```

Parameters

- config – Pointer to the onfiguration structure.

static inline uint32_t RIT_GetStatusFlags(RIT_Type *base)

> Gets the RIT status flags.

Parameters

- base – RIT peripheral base address

Returns

> The status flags. This is the logical OR of members of the enumeration rit_status_flags_t

static inline void RIT_ClearStatusFlags(RIT_Type *base, uint32_t mask)

> Clears the RIT status flags.

Parameters

- base – RIT peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration rit_status_flags_t

void RIT_SetTimerCompare(RIT_Type *base, uint64_t count)

> Sets the timer period in units of count.
>
> This function sets the RI compare value. If the counter value equals to the compare value, it will generate an interrupt.

Note: Users can call the utility macros provided in fsl_common.h to convert to ticks

---

Parameters

- base – RIT peripheral base address

- count – Timer period in units of ticks

void RIT_SetMaskBit(RIT_Type *base, uint64_t count)

Sets the mask bit of count compare.

This function sets the RI mask value. A 1 written to any bit will force the compare to be true for the corresponding bit of the counter and compare register (causes the comparison of the register bits to be always true).

Note: Users can call the utility macros provided in fsl_common.h to convert to ticks

---

Parameters

- base – RIT peripheral base address

- count – Timer period in units of ticks

uint64_t RIT_GetCompareTimerCount(RIT_Type *base)

Reads the current value of compare register.

Note: Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

---

Parameters

- base – RIT peripheral base address

Returns

Current RI compare value

uint64_t RIT_GetCounterTimerCount(RIT_Type *base)

Reads the current timer counting value of counter register.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

---

Parameters

- base – RIT peripheral base address

Returns

Current timer counting value in ticks

uint64_t RIT_GetMaskTimerCount(RIT_Type *base)

Reads the current value of mask register.

Note: Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

---

Parameters

- base – RIT peripheral base address

Returns

Current RI mask value

---

static inline void RIT_StartTimer(RIT_Type *base)

> Starts the timer counting.

> After calling this function, timers load initial value(0U), count up to desired value or overflow then the counter will count up again.

> **Parameters**

>> • base – RIT peripheral base address

static inline void RIT_StopTimer(RIT_Type *base)

> Stops the timer counting.

> This function stop timer counting. Timer reload their new value after the next time they call the RIT_StartTimer.

> **Parameters**

>> • base – RIT peripheral base address

FSL_RIT_DRIVER_VERSION

> Version 2.1.2

enum __rit_status_flags

> List of RIT status flags.

> *Values:*

> enumerator kRIT_TimerFlag

>> Timer flag

typedef enum _rit_status_flags rit_status_flags_t

> List of RIT status flags.

typedef struct _rit_config rit_config_t

> RIT config structure.

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

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

static inline void RIT_ClearCounter(RIT_Type *base, bool enable)

> Sets the Timer Counter auto clear or not.

> This function set the counter auto clear or not whenever the counter value equals the masked compare value specified by the contents of COMPVAL/COMPVAL_H and MASK/MASK_H registers..

> *Deprecated:*

>> Do not use this function. It has been superceded by RIT_SetCountAutoClear.

static inline void RIT_SetCountAutoClear(RIT_Type *base, bool enable)

> Sets the Timer Counter auto clear or not.

> This function set the counter auto clear or not whenever the counter value equals the masked compare value specified by the contents of COMPVAL/COMPVAL_H and MASK/MASK_H registers..

> **Parameters**

>> • base – RIT peripheral base address

>> • enable – Enable/disable Counter auto clear when value equals the compare value.

2.37. RIT: Repetitive Interrupt Timer

– true: Enable Counter auto clear.

– false: Disable Counter auto clear.

struct __rit_config

#include <fsl_rit.h> RIT config structure.

This structure holds the configuration settings for the RIT peripheral. To initialize this structure to reasonable defaults, call the RIT_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 enableRunInDebug

true: The timer is halted when the processor is halted for debugging.; false: Debug has no effect on the timer operation.

2.38 RNG: Random Number Generator

FSL_RNG_DRIVER_VERSION

RNG driver version 2.1.0.

Current version: 2.1.0

Change log:

- Version 2.0.0

    – Initial version.

- Version 2.1.0

    – Renamed function RNG_GetRandomData() to RNG_GetRandomDataWord(). Added function RNG_GetRandomData() which discarding next 32 words after reading RNG register which results into better entropy, as is recommended in UM.

    – API is aligned with other RNG driver, having similar functionality as other RNG/TRNG drivers.

status_t RNG_GetRandomData(void *data, size_t dataSize)

Gets random data.

This function gets random data from the RNG.

Parameters

- data – Pointer address used to store random data.

- dataSize – Size of the buffer pointed by the data parameter.

Returns

Status from operation

static inline uint32_t RNG_GetRandomDataWord(void)

Gets random data.

This function returns single 32 bit random number. For each read word next 32 words should be discarded to achieve better entropy.

Returns

random data

FSL_COMPONENT_ID

2.39   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 independly, but they can't use the RTC wake-up timer (1KHZ) independently.

Parameters

- base – RTC peripheral base address

- enable – Enable/Disable RTC Timer counter.

  – true: Enable RTC Timer counter.

  – false: Disable RTC Timer counter.

static inline void RTC_StartTimer(RTC_Type *base)

Starts the RTC time counter.

Deprecated:

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

After calling this function, the timer counter increments once a second provided SR[TOF] or SR[TIF] are not set.

Parameters

- base – RTC peripheral base address

static inline void RTC_StopTimer(RTC_Type *base)

Stops the RTC time counter.

Deprecated:

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

RTC's seconds register can be written to only when the timer is stopped.

Parameters

- base – RTC peripheral base address

FSL_RTC_DRIVER_VERSION

Version 2.2.0

enum _rtc_interrupt_enable

List of RTC interrupts.

Values:

enumerator kRTC_AlarmInterruptEnable

Alarm interrupt.

enumerator kRTC_WakeupInterruptEnable

Wake-up interrupt.

enum _rtc_status_flags

List of RTC flags.

Values:

enumerator kRTC_AlarmFlag

Alarm flag

enumerator kRTC_WakeupFlag

1kHz wake-up timer flag

typedef enum *_rtc_interrupt_enable* rtc_interrupt_enable_t

List of RTC interrupts.

typedef enum *_rtc_status_flags* rtc_status_flags_t
> List of RTC flags.

typedef struct *_rtc_datetime* rtc_datetime_t
> Structure is used to hold the date and time.

static inline void RTC_SetSecondsTimerMatch(RTC_Type *base, uint32_t matchValue)
> Set the RTC seconds timer (1HZ) MATCH value.

> > **Parameters**
> > - base – RTC peripheral base address
> > - matchValue – The value to be set into the RTC MATCH register

static inline uint32_t RTC_GetSecondsTimerMatch(RTC_Type *base)
> Read actual RTC seconds timer (1HZ) MATCH value.

> > **Parameters**
> > - base – RTC peripheral base address

> > **Returns**
> > The actual RTC seconds timer (1HZ) MATCH value.

static inline void RTC_SetSecondsTimerCount(RTC_Type *base, uint32_t countValue)
> Set the RTC seconds timer (1HZ) COUNT value.

> > **Parameters**
> > - base – RTC peripheral base address
> > - countValue – The value to be loaded into the RTC COUNT register

static inline uint32_t RTC_GetSecondsTimerCount(RTC_Type *base)
> Read the actual RTC seconds timer (1HZ) COUNT value.

> > **Parameters**
> > - base – RTC peripheral base address

> > **Returns**
> > The actual RTC seconds timer (1HZ) COUNT value.

static inline void RTC_SetWakeupCount(RTC_Type *base, uint16_t wakeupValue)
> Enable the RTC wake-up timer (1KHZ) and set countdown value to the RTC WAKE register.

> > **Parameters**
> > - base – RTC peripheral base address
> > - wakeupValue – The value to be loaded into the WAKE register in RTC wake-up timer (1KHZ).

static inline uint16_t RTC_GetWakeupCount(RTC_Type *base)
> Read the actual value from the WAKE register value in RTC wake-up timer (1KHZ)

> Read the WAKE register twice and compare the result, if the value match,the time can be used.

> > **Parameters**
> > - base – RTC peripheral base address

> > **Returns**
> > The actual value of the WAKE register value in RTC wake-up timer (1KHZ).

static inline void RTC_Reset(RTC_Type *base)

> Perform a software reset on the RTC module.
>
> This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.
>
> > **Parameters**
> >
> > > • base – RTC peripheral base address

struct __rtc_datetime

> *#include <fsl_rtc.h>* Structure is used to hold the date and time.

> ### Public Members

> uint16_t year
>
> > Range from 1970 to 2099.

> uint8_t month
>
> > Range from 1 to 12.

> uint8_t day
>
> > Range from 1 to 31 (depending on month).

> uint8_t hour
>
> > Range from 0 to 23.

> uint8_t minute
>
> > Range from 0 to 59.

> uint8_t second
>
> > Range from 0 to 59.

2.40 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 dutycylce 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 SC-TIMER_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 noticification 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 SC-TIMER_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__STATEMSKn(**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 INPUTSYNC0 (0U) #define INPUTSYNC1 (1U) #define INPUTSYNC2 (2U) #define INPUTSYNC3 (3U) User Code. sctimerInfo.inputsync = (1 « INPUTSYNC2) | (1 « INPUTSYNC3);

2.41 SDIF: SD/MMC/SDIO card interface

FSL__SDIF__DRIVER__VERSION

Driver version 2.0.15.

_sdif_status SDIF status

Values:

enumerator kStatus_SDIF__DescriptorBufferLenError

Set DMA descriptor failed

enumerator kStatus_SDIF__InvalidArgument

invalid argument status

enumerator kStatus_SDIF__SyncCmdTimeout

sync command to CIU timeout status

enumerator kStatus_SDIF__SendCmdFail

send command to card fail

enumerator kStatus_SDIF__SendCmdErrorBufferFull

send command to card fail, due to command buffer full user need to resend this command

enumerator kStatus_SDIF__DMATransferFailWithFBE

DMA transfer data fail with fatal bus error , to do with this error :issue a hard reset/controller reset

enumerator kStatus_SDIF__DMATransferDescriptorUnavailable

DMA descriptor unavailable

enumerator kStatus_SDIF__DataTransferFail

transfer data fail

enumerator kStatus_SDIF_ResponseError
    response error

enumerator kStatus_SDIF_DMAAddrNotAlign
    DMA address not align

enumerator kStatus_SDIF_BusyTransferring
    SDIF transfer busy status

enumerator kStatus_SDIF_DataTransferSuccess
    transfer data success

enumerator kStatus_SDIF_SendCmdSuccess
    transfer command success

_sdif_capability_flag Host controller capabilities flag mask

Values:

enumerator kSDIF_SupportHighSpeedFlag
    Support high-speed

enumerator kSDIF_SupportDmaFlag
    Support DMA

enumerator kSDIF_SupportSuspendResumeFlag
    Support suspend/resume

enumerator kSDIF_SupportV330Flag
    Support voltage 3.3V

enumerator kSDIF_Support4BitFlag
    Support 4 bit mode

enumerator kSDIF_Support8BitFlag
    Support 8 bit mode

_sdif_reset_type define the reset type

Values:

enumerator kSDIF_ResetController
    reset controller,will reset: BIU/CIU interface CIU and state machine,ABORT_READ_DATA,SEND_IRQ_RESPONSE and READ_WAIT bits of control register,START_CMD bit of the command register

enumerator kSDIF_ResetFIFO
    reset data FIFO

enumerator kSDIF_ResetDMAInterface
    reset DMA interface

enumerator kSDIF_ResetAll
    reset all

enum _sdif_bus_width
    define the card bus width type

    Values:

    enumerator kSDIF_Bus1BitWidth
        1bit bus width, 1bit mode and 4bit mode share one register bit

enumerator kSDIF_Bus4BitWidth
    4bit mode mask

enumerator kSDIF_Bus8BitWidth
    support 8 bit mode

_sdif_command_flags define the command flags

Values:

enumerator kSDIF_CmdResponseExpect
    command request response

enumerator kSDIF_CmdResponseLengthLong
    command response length long

enumerator kSDIF_CmdCheckResponseCRC
    request check command response CRC

enumerator kSDIF_DataExpect
    request data transfer,either read/write

enumerator kSDIF_DataWriteToCard
    data transfer direction

enumerator kSDIF_DataStreamTransfer
    data transfer mode :stream/block transfer command

enumerator kSDIF_DataTransferAutoStop
    data transfer with auto stop at the end of

enumerator kSDIF_WaitPreTransferComplete
    wait pre transfer complete before sending this cmd

enumerator kSDIF_TransferStopAbort
    when host issue stop or abort cmd to stop data transfer ,this bit should set so that cmd/data state-machines of CIU can return to idle correctly

enumerator kSDIF_SendInitialization
    send initialization 80 clocks for SD card after power on

enumerator kSDIF_CmdUpdateClockRegisterOnly
    send cmd update the CIU clock register only

enumerator kSDIF_CmdtoReadCEATADevice
    host is perform read access to CE-ATA device

enumerator kSDIF_CmdExpectCCS
    command expect command completion signal signal

enumerator kSDIF_BootModeEnable
    this bit should only be set for mandatory boot mode

enumerator kSDIF_BootModeExpectAck
    boot mode expect ack

enumerator kSDIF_BootModeDisable
    when software set this bit along with START_CMD, CIU terminates the boot operation

enumerator kSDIF_BootModeAlternate
    select boot mode ,alternate or mandatory

enumerator kSDIF_CmdVoltageSwitch
    this bit set for CMD11 only

enumerator kSDIF_CmdDataUseHoldReg
    cmd and data send to card through the HOLD register

_sdif_command_type The command type

Values:

enumerator kCARD_CommandTypeNormal
    Normal command

enumerator kCARD_CommandTypeSuspend
    Suspend command

enumerator kCARD_CommandTypeResume
    Resume command

enumerator kCARD_CommandTypeAbort
    Abort command

_sdif_response_type The command response type.

Define the command response type from card to host controller.

Values:

enumerator kCARD_ResponseTypeNone
    Response type: none

enumerator kCARD_ResponseTypeR1
    Response type: R1

enumerator kCARD_ResponseTypeR1b
    Response type: R1b

enumerator kCARD_ResponseTypeR2
    Response type: R2

enumerator kCARD_ResponseTypeR3
    Response type: R3

enumerator kCARD_ResponseTypeR4
    Response type: R4

enumerator kCARD_ResponseTypeR5
    Response type: R5

enumerator kCARD_ResponseTypeR5b
    Response type: R5b

enumerator kCARD_ResponseTypeR6
    Response type: R6

enumerator kCARD_ResponseTypeR7
    Response type: R7

_sdif_interrupt_mask define the interrupt mask flags

Values:

enumerator kSDIF_CardDetect

mask for card detect

enumerator kSDIF_ResponseError

command response error

enumerator kSDIF_CommandDone

command transfer over

enumerator kSDIF_DataTransferOver

data transfer over flag

enumerator kSDIF_WriteFIFORequest

write FIFO request

enumerator kSDIF_ReadFIFORequest

read FIFO request

enumerator kSDIF_ResponseCRCError

response CRC error

enumerator kSDIF_DataCRCError

data CRC error

enumerator kSDIF_ResponseTimeout

response timeout

enumerator kSDIF_DataReadTimeout

read data timeout

enumerator kSDIF_DataStarvationByHostTimeout

data starvation by host time out

enumerator kSDIF_FIFOError

indicate the FIFO under run or overrun error

enumerator kSDIF_HardwareLockError

hardware lock write error

enumerator kSDIF_DataStartBitError

start bit error

enumerator kSDIF_AutoCmdDone

indicate the auto command done

enumerator kSDIF_DataEndBitError

end bit error

enumerator kSDIF_SDIOInterrupt

interrupt from the SDIO card

enumerator kSDIF_CommandTransferStatus

command transfer status collection

enumerator kSDIF_DataTransferStatus

data transfer status collection

enumerator kSDIF_DataTransferError

enumerator kSDIF_AllInterruptStatus

all interrupt mask

_sdif_dma_status define the internal DMA status flags

Values:

enumerator kSDIF_DMATransFinishOneDescriptor
    DMA transfer finished for one DMA descriptor

enumerator kSDIF_DMARecvFinishOneDescriptor
    DMA receive finished for one DMA descriptor

enumerator kSDIF_DMAFatalBusError
    DMA fatal bus error

enumerator kSDIF_DMADescriptorUnavailable
    DMA descriptor unavailable

enumerator kSDIF_DMACardErrorSummary
    card error summary

enumerator kSDIF_NormalInterruptSummary
    normal interrupt summary

enumerator kSDIF_AbnormalInterruptSummary
    abnormal interrupt summary

enumerator kSDIF_DMAAllStatus

_sdif_dma_descriptor_flag define the internal DMA descriptor flag

Deprecated:
    Do not use this enum anymore, please use SDIF_DMA_DESCRIPTOR_XXX_FLAG instead.

Values:

enumerator kSDIF_DisableCompleteInterrupt
    disable the complete interrupt flag for the ends in the buffer pointed to by this descriptor

enumerator kSDIF_DMADescriptorDataBufferEnd
    indicate this descriptor contain the last data buffer of data

enumerator kSDIF_DMADescriptorDataBufferStart
    indicate this descriptor contain the first data buffer of data,if first buffer size is 0,next descriptor contain the begin of the data

enumerator kSDIF_DMASecondAddrChained
    indicate that the second addr in the descriptor is the next descriptor addr not the data buffer

enumerator kSDIF_DMADescriptorEnd
    indicate that the descriptor list reached its final descriptor

enumerator kSDIF_DMADescriptorOwnByDMA
    indicate the descriptor is own by SD/MMC DMA

enum _sdif_dma_mode
    define the internal DMA mode

    Values:

enumerator kSDIF_ChainDMAMode

enumerator kSDIF_DualDMAMode

typedef enum _*sdif_bus_width* sdif_bus_width_t
    define the card bus width type

typedef enum _*sdif_dma_mode* sdif_dma_mode_t
    define the internal DMA mode

typedef struct _*sdif_dma_descriptor* sdif_dma_descriptor_t
    define the internal DMA descriptor

typedef struct _*sdif_dma_config* sdif_dma_config_t
    Defines the internal DMA configure structure.

typedef struct _*sdif_data* sdif_data_t
    Card data descriptor.

typedef struct _*sdif_command* sdif_command_t
    Card command descriptor.

    Define card command-related attribute.

typedef struct _*sdif_transfer* sdif_transfer_t
    Transfer state.

typedef struct _*sdif_config* sdif_config_t
    Data structure to initialize the sdif.

typedef struct _*sdif_capability* sdif_capability_t
    SDIF capability information. Defines a structure to get the capability information of SDIF.

typedef struct _*sdif_transfer_callback* sdif_transfer_callback_t
    sdif callback functions.

typedef struct _*sdif_handle* sdif_handle_t
    sdif handle

    Defines the structure to save the sdif state information and callback function. The detail interrupt status when send command or transfer data can be obtained from interruptFlags field by using mask defined in sdif_interrupt_flag_t;

    Note: All the fields except interruptFlags and transferredWords must be allocated by the user.

    ---

typedef *status_t* (*sdif_transfer_function_t)(SDIF_Type *base, *sdif_transfer_t* *content)
    sdif transfer function.

typedef struct _*sdif_host* sdif_host_t
    sdif host descriptor

void SDIF_Init(SDIF_Type *base, *sdif_config_t* *config)
    SDIF module initialization function.

    Configures the SDIF according to the user configuration.

        Parameters

            • base – SDIF peripheral base address.

            • config – SDIF configuration information.

void SDIF_Deinit(SDIF_Type *base)

    SDIF module deinit function. user should call this function follow with IP reset.

        Parameters

                • base – SDIF peripheral base address.

bool SDIF_SendCardActive(SDIF_Type *base, uint32_t timeout)

    SDIF send initialize 80 clocks for SD card after initial.

        Parameters

                • base – SDIF peripheral base address.

                • timeout – timeout value

static inline void SDIF_EnableCardClock(SDIF_Type *base, bool enable)

    SDIF module enable/disable card clock.

        Parameters

                • base – SDIF peripheral base address.

                • enable – enable/disable flag

static inline void SDIF_EnableLowPowerMode(SDIF_Type *base, bool enable)

    SDIF module enable/disable module disable the card clock to enter low power mode when card is idle,for SDIF cards, if interrupts must be detected, clock should not be stopped.

        Parameters

                • base – SDIF peripheral base address.

                • enable – enable/disable flag

static inline void SDIF_EnableCardPower(SDIF_Type *base, bool enable)

    enable/disable the card power. once turn power on, software should wait for regulator/switch ramp-up time before trying to initialize card.

        Parameters

                • base – SDIF peripheral base address.

                • enable – enable/disable flag.

void SDIF_SetCardBusWidth(SDIF_Type *base, *sdif_bus_width_t* type)

    set card data bus width

        Parameters

                • base – SDIF peripheral base address.

                • type – bus width type

static inline uint32_t SDIF_DetectCardInsert(SDIF_Type *base, bool data3)

    SDIF module detect card insert status function.

        Parameters

                • base – SDIF peripheral base address.

                • data3 – indicate use data3 as card insert detect pin

        Return values

            1 – card is inserted 0 card is removed

uint32_t SDIF_SetCardClock(SDIF_Type *base, uint32_t srcClock_Hz, uint32_t target_HZ)

    Sets the card bus clock frequency.

        Parameters

- base – SDIF peripheral base address.

- srcClock_Hz – SDIF source clock frequency united in Hz.

- target_HZ – card bus clock frequency united in Hz.

> **Returns**
> The nearest frequency of busClock_Hz configured to SD bus.

bool SDIF_Reset(SDIF_Type *base, uint32_t mask, uint32_t timeout)

> reset the different block of the interface.

> **Parameters**

- base – SDIF peripheral base address.

- mask – indicate which block to reset.

- timeout – timeout value,set to wait the bit self clear

> **Returns**
> reset result.

static inline uint32_t SDIF_GetCardWriteProtect(SDIF_Type *base)

> get the card write protect status

> **Parameters**

- base – SDIF peripheral base address.

static inline void SDIF_AssertHardwareReset(SDIF_Type *base)

> toggle state on hardware reset PIN This is used which card has a reset PIN typically.

> **Parameters**

- base – SDIF peripheral base address.

status_t SDIF_SendCommand(SDIF_Type *base, *sdif_command_t* *cmd, uint32_t timeout)

> send command to the card

> This api include polling the status of the bit START_COMMAND, if 0 used as timeout value, then this function will return directly without polling the START_CMD status.

> **Parameters**

- base – SDIF peripheral base address.

- cmd – configuration collection

- timeout – the timeout value of polling START_CMD auto clear status.

> **Returns**
> command excute status

static inline void SDIF_EnableGlobalInterrupt(SDIF_Type *base, bool enable)

> SDIF enable/disable global interrupt.

> **Parameters**

- base – SDIF peripheral base address.

- enable – enable/disable flag

static inline void SDIF_EnableInterrupt(SDIF_Type *base, uint32_t mask)

> SDIF enable interrupt.

> **Parameters**

- base – SDIF peripheral base address.

- mask – mask

static inline void SDIF_DisableInterrupt(SDIF_Type *base, uint32_t mask)

SDIF disable interrupt.

Parameters

- base – SDIF peripheral base address.

- mask – mask

static inline uint32_t SDIF_GetInterruptStatus(SDIF_Type *base)

SDIF get interrupt status.

Parameters

- base – SDIF peripheral base address.

static inline uint32_t SDIF_GetEnabledInterruptStatus(SDIF_Type *base)

SDIF get enabled interrupt status.

Parameters

- base – SDIF peripheral base address.

static inline void SDIF_ClearInterruptStatus(SDIF_Type *base, uint32_t mask)

SDIF clear interrupt status.

Parameters

- base – SDIF peripheral base address.

- mask – mask to clear

void SDIF_TransferCreateHandle(SDIF_Type *base, *sdif_handle_t* *handle,
                               sdif_transfer_callback_t *callback, void *userData)

Creates the SDIF handle. register call back function for interrupt and enable the interrupt.

Parameters

- base – SDIF peripheral base address.

- handle – SDIF handle pointer.

- callback – Structure pointer to contain all callback functions.

- userData – Callback function parameter.

static inline void SDIF_EnableDmaInterrupt(SDIF_Type *base, uint32_t mask)

SDIF enable DMA interrupt.

Parameters

- base – SDIF peripheral base address.

- mask – mask to set

static inline void SDIF_DisableDmaInterrupt(SDIF_Type *base, uint32_t mask)

SDIF disable DMA interrupt.

Parameters

- base – SDIF peripheral base address.

- mask – mask to clear

static inline uint32_t SDIF_GetInternalDMAStatus(SDIF_Type *base)

SDIF get internal DMA status.

Parameters

- base – SDIF peripheral base address.

Returns

the internal DMA status register

static inline uint32_t SDIF_GetEnabledDMAInterruptStatus(SDIF_Type *base)

SDIF get enabled internal DMA interrupt status.

Parameters

- base – SDIF peripheral base address.

Returns

the internal DMA status register

static inline void SDIF_ClearInternalDMAStatus(SDIF_Type *base, uint32_t mask)

SDIF clear internal DMA status.

Parameters

- base – SDIF peripheral base address.

- mask – mask to clear

status_t SDIF_InternalDMAConfig(SDIF_Type *base, *sdif_dma_config_t* *config, const uint32_t *data, uint32_t dataSize)

SDIF internal DMA config function.

Parameters

- base – SDIF peripheral base address.

- config – DMA configuration collection

- data – buffer pointer

- dataSize – buffer size

static inline void SDIF_EnableInternalDMA(SDIF_Type *base, bool enable)

SDIF internal DMA enable.

Parameters

- base – SDIF peripheral base address.

- enable – internal DMA enable or disable flag.

static inline void SDIF_SendReadWait(SDIF_Type *base)

SDIF send read wait to SDIF card function.

Parameters

- base – SDIF peripheral base address.

bool SDIF_AbortReadData(SDIF_Type *base, uint32_t timeout)

SDIF abort the read data when SDIF card is in suspend state Once assert this bit,data state machine will be reset which is waiting for the next blocking data,used in SDIO card suspend sequence,should call after suspend cmd send.

Parameters

- base – SDIF peripheral base address.

- timeout – timeout value to wait this bit self clear which indicate the data machine reset to idle

static inline void SDIF_EnableCEATAInterrupt(SDIF_Type *base, bool enable)

SDIF enable/disable CE-ATA card interrupt this bit should set together with the card register.

Parameters

- base – SDIF peripheral base address.

- enable – enable/disable flag

status_t SDIF__TransferNonBlocking(SDIF_Type *base, *sdif_handle_t* *handle, *sdif_dma_config_t* *dmaConfig, *sdif_transfer_t* *transfer)

SDIF transfer function data/cmd in a non-blocking way this API should be use in interrupt mode, when use this API user must call SDIF_TransferCreateHandle first, all status check through interrupt.

Parameters

- base – SDIF peripheral base address.

- handle – handle

- dmaConfig – config structure This parameter can be config as:

  a. NULL In this condition, polling transfer mode is selected

  b. avaliable DMA config In this condition, DMA transfer mode is selected

- transfer – transfer configuration collection

status_t SDIF__TransferBlocking(SDIF_Type *base, *sdif_dma_config_t* *dmaConfig, *sdif_transfer_t* *transfer)

SDIF transfer function data/cmd in a blocking way.

Parameters

- base – SDIF peripheral base address.

- dmaConfig – config structure

  a. NULL In this condition, polling transfer mode is selected

  b. avaliable DMA config In this condition, DMA transfer mode is selected

- transfer – transfer configuration collection

status_t SDIF__ReleaseDMADescriptor(SDIF_Type *base, *sdif_dma_config_t* *dmaConfig)

SDIF release the DMA descriptor to DMA engine this function should be called when DMA descriptor unavailable status occurs.

Parameters

- base – SDIF peripheral base address.

- dmaConfig – DMA config pointer

void SDIF__GetCapability(SDIF_Type *base, *sdif_capability_t* *capability)

SDIF return the controller capability.

Parameters

- base – SDIF peripheral base address.

- capability – capability pointer

static inline uint32_t SDIF__GetControllerStatus(SDIF_Type *base)

SDIF return the controller status.

Parameters

- base – SDIF peripheral base address.

static inline void SDIF__SendCCSD(SDIF_Type *base, bool withAutoStop)

SDIF send command complete signal disable to CE-ATA card.

Parameters

- base – SDIF peripheral base address.

- withAutoStop – auto stop flag

void SDIF_ConfigClockDelay(uint32_t target_HZ, uint32_t divider)

> SDIF config the clock delay This function is used to config the cclk_in delay to sample and driver the data ,should meet the min setup time and hold time, and user need to config this parameter according to your board setting.

> > **Parameters**

> > > • target_HZ – freq work mode

> > > • divider – not used in this function anymore, use DELAY value instead of phase directly.

SDIF_CLOCK_RANGE_NEED_DELAY

> SDIOCLKCTRL setting Below clock delay setting should depend on specific platform, so it can be redefined when timing mismatch issue occur. Such as: response error/CRC error and so on.

> clock range value which need to add delay to avoid timing issue

SDIF_HIGHSPEED_SAMPLE_DELAY

> High speed mode clk_sample fixed delay.

> 12 * 250ps = 3ns

SDIF_HIGHSPEED_DRV_DELAY

> High speed mode clk_drv fixed delay.

> 31 * 250ps = 7.75ns

SDIF_HIGHSPEED_SAMPLE_PHASE_SHIFT

> High speed mode clk_sample phase shift.

SDIF_HIGHSPEED_DRV_PHASE_SHIFT

> High speed mode clk_drv phase shift.

SDIF_DEFAULT_MODE_SAMPLE_DELAY

> default mode sample fixed delay

> 12 * 250ps = 3ns

SDIF_DEFAULT_MODE_DRV_DELAY

> 31 * 250ps = 7.75ns

SDIF_INTERNAL_DMA_ADDR_ALIGN

> SDIF internal DMA descriptor address and the data buffer address align.

SDIF_DMA_DESCRIPTOR_DISABLE_COMPLETE_INT_FLAG

> SDIF DMA descriptor flag.

SDIF_DMA_DESCRIPTOR_DATA_BUFFER_END_FLAG

SDIF_DMA_DESCRIPTOR_DATA_BUFFER_START_FLAG

SDIF_DMA_DESCRIPTOR_SECOND_ADDR_CHAIN_FLAG

SDIF_DMA_DESCRIPTOR_DESCRIPTOR_END_FLAG

SDIF_DMA_DESCRIPTOR_OWN_BY_DMA_FLAG

struct __sdif_dma_descriptor

> *#include <fsl_sdif.h>* define the internal DMA descriptor

Public Members

uint32_t dmaDesAttribute

internal DMA attribute control and status

uint32_t dmaDataBufferSize

internal DMA transfer buffer size control

const uint32_t *dmaDataBufferAddr0

internal DMA buffer 0 addr ,the buffer size must be 32bit aligned

const uint32_t *dmaDataBufferAddr1

internal DMA buffer 1 addr ,the buffer size must be 32bit aligned

struct __sdif__dma__config

#include <fsl_sdif.h> Defines the internal DMA configure structure.

Public Members

bool enableFixBurstLen

fix burst len enable/disable flag,When set, the AHB will use only SINGLE, INCR4, INCR8 or INCR16 during start of normal burst transfers. When reset, the AHB will use SINGLE and INCR burst transfer operations

sdif_dma_mode_t mode

define the DMA mode

uint8_t dmaDesSkipLen

define the descriptor skip length ,the length between two descriptor this field is special for dual DMA mode

uint32_t *dmaDesBufferStartAddr

internal DMA descriptor start address

uint32_t dmaDesBufferLen

internal DMA buffer descriptor buffer len ,user need to pay attention to the dma descriptor buffer length if it is bigger enough for your transfer

struct __sdif__data

#include <fsl_sdif.h> Card data descriptor.

Public Members

bool streamTransfer

indicate this is a stream data transfer command

bool enableAutoCommand12

indicate if auto stop will send when data transfer over

bool enableIgnoreError

indicate if enable ignore error when transfer data

size_t blockSize

Block size, take care when configure this parameter

uint32_t blockCount

Block count

uint32_t *rxData

data buffer to receive

const uint32_t *txData
    data buffer to transfer

struct __sdif_command
    #include <fsl_sdif.h> Card command descriptor.

    Define card command-related attribute.

Public Members

uint32_t index
    Command index

uint32_t argument
    Command argument

uint32_t response[4U]
    Response for this command

uint32_t type
    define the command type

uint32_t responseType
    Command response type

uint32_t flags
    Cmd flags

uint32_t responseErrorFlags
    response error flags, need to check the flags when receive the cmd response

struct __sdif_transfer
    #include <fsl_sdif.h> Transfer state.

Public Members

sdif_data_t *data
    Data to transfer

sdif_command_t *command
    Command to send

struct __sdif_config
    #include <fsl_sdif.h> Data structure to initialize the sdif.

Public Members

uint8_t responseTimeout
    command response timeout value

uint32_t cardDetDebounce_Clock
    define the debounce clock count which will used in card detect logic,typical value is 5-25ms

uint32_t dataTimeout
    data timeout value

struct __sdif_capability
    #include <fsl_sdif.h> SDIF capability information. Defines a structure to get the capability information of SDIF.

2.41. SDIF: SD/MMC/SDIO card interface   **415**

Public Members

uint32_t sdVersion
    support SD card/sdio version

uint32_t mmcVersion
    support emmc card version

uint32_t maxBlockLength
    Maximum block length united as byte

uint32_t maxBlockCount
    Maximum byte count can be transfered

uint32_t flags
    Capability flags to indicate the support information

struct __sdif_transfer_callback
    #include <fsl_sdif.h> sdif callback functions.

Public Members

void (*cardInserted)(SDIF_Type *base, void *userData)
    card insert call back

void (*cardRemoved)(SDIF_Type *base, void *userData)
    card remove call back

void (*SDIOInterrupt)(SDIF_Type *base, void *userData)
    SDIO card interrupt occurs

void (*DMADesUnavailable)(SDIF_Type *base, void *userData)
    DMA descriptor unavailable

void (*CommandReload)(SDIF_Type *base, void *userData)
    command buffer full,need re-load

void (*TransferComplete)(SDIF_Type *base, void *handle, *status_t* status, void *userData)
    Transfer complete callback

struct __sdif_handle
    #include <fsl_sdif.h> sdif handle

    Defines the structure to save the sdif state information and callback function. The detail interrupt status when send command or transfer data can be obtained from interruptFlags field by using mask defined in sdif_interrupt_flag_t;

Note: All the fields except interruptFlags and transferredWords must be allocated by the user.

Public Members

sdif_data_t *volatile data
    Data to transfer

sdif_command_t *volatile command
    Command to send

volatile uint32_t transferredWords

Words transferred by polling way

sdif_transfer_callback_t callback

Callback function

void *userData

Parameter for transfer complete callback

struct __sdif_host

#include <fsl_sdif.h> sdif host descriptor

Public Members

SDIF_Type *base

sdif peripheral base address

uint32_t sourceClock_Hz

sdif source clock frequency united in Hz

sdif_config_t config

sdif configuration

sdif_transfer_function_t transfer

sdif transfer function

sdif_capability_t capability

sdif capability information

2.42   SHA: SHA encryption decryption driver

FSL_SHA_DRIVER_VERSION

Defines LPC SHA driver version 2.3.2.

Current version: 2.3.2

Change log:

- Version 2.0.0

    - Initial version

- Version 2.1.0

    - Updated "sha_ldm_stm_16_words" "sha_one_block" API to match QN9090. QN9090 has no ALIAS register.

    - Added "SHA_ClkInit" "SHA_ClkInit"

- Version 2.1.1

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

- Version 2.2.0

    - Support MEMADDR pseudo DMA for loading input data in SHA_Update function (LPCXpresso54018 and LPCXpresso54628).

- Version 2.2.1

    - MISRA C-2012 issue fix.

- Version 2.2.2 Modified SHA_Finish function. While using pseudo DMA with maximum optimization, compiler optimize out condition. Which caused block in this function and did not check state, which has been set in interrupt.

- Version 2.3.0 Modified SHA_Update to use blocking version of AHB Master mode when its available on chip. Added SHA_UpdateNonBlocking() function which uses nonblocking AHB Master mode. Fixed incorrect calculation of SHA when calling SHA_Update multiple times when is CPU used to load data. Added Reset into SHA_ClkInit and SHA_ClkDeinit function.

- Version 2.3.1 Modified sha_process_message_data_master() to ensure that MEMCTRL will be written within 64 cycles of writing last word to INDATA as is mentioned in errata, even with different optimization levels.

- Version 2.3.2 Add -O2 optimization for GCC to sha_process_message_data_master(), because without it the function hangs under some conditions.

enum _sha_algo_t

Supported cryptographic block cipher functions for HASH creation

Values:

enumerator kSHA_Sha1

SHA_1

enumerator kSHA_Sha256

SHA_256

typedef enum *_sha_algo_t* sha_algo_t

Supported cryptographic block cipher functions for HASH creation

typedef struct *_sha_ctx_t* sha_ctx_t

Storage type used to save hash context.

typedef void (*sha_callback_t)(SHA_Type *base, *sha_ctx_t* *ctx, *status_t* status, void *userData)

background hash callback function.

SHA_CTX_SIZE

SHA Context size.

struct _sha_ctx_t

#include <fsl_sha.h> Storage type used to save hash context.

2.43 Sha_algorithm_level_api

status_t SHA_Init(SHA_Type *base, *sha_ctx_t* *ctx, *sha_algo_t* algo)

Initialize HASH context.

This function initializes new hash context.

Parameters

- base – SHA peripheral base address

- ctx – **[out]** Output hash context

- algo – Underlaying algorithm to use for hash computation. Either SHA-1 or SHA-256.

Returns

Status of initialization

status_t SHA__Update(SHA_Type *base, *sha_ctx_t* *ctx, const uint8_t *message, size_t
            messageSize)

Add data to current HASH.

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data
to be hashed.

Parameters

- base – SHA peripheral base address

- ctx – **[inout]** HASH context

- message – Input message

- messageSize – Size of input message in bytes

Returns

Status of the hash update operation

status_t SHA__Finish(SHA_Type *base, *sha_ctx_t* *ctx, uint8_t *output, size_t *outputSize)

Finalize hashing.

Outputs the final hash and erases the context. SHA-1 or SHA-256 padding bits are automat-
ically added by this function.

Parameters

- base – SHA peripheral base address

- ctx – **[inout]** HASH context

- output – **[out]** Output hash data

- outputSize – **[inout]** On input, determines the size of bytes of the output
  array. On output, tells how many bytes have been written to output.

Returns

Status of the hash finish operation

void SHA__SetCallback(SHA_Type *base, *sha_ctx_t* *ctx, *sha_callback_t* callback, void *userData)

Initializes the SHA handle for background hashing.

This function initializes the hash context for background hashing (Non-blocking) APIs. This
is less typical interface to hash function, but can be used for parallel processing, when
main CPU has something else to do. Example is digital signature RSASSA-PKCS1-V1_5-
VERIFY((n,e),M,S) algorithm, where background hashing of M can be started, then CPU can
compute S^e mod n (in parallel with background hashing) and once the digest becomes
available, CPU can proceed to comparison of EM with EM'.

Parameters

- base – SHA peripheral base address.

- ctx – **[out]** Hash context.

- callback – Callback function.

- userData – User data (to be passed as an argument to callback function,
  once callback is invoked from isr).

status_t SHA__UpdateNonBlocking(SHA_Type *base, *sha_ctx_t* *ctx, const uint8_t *input, size_t
                    inputSize)

Create running hash on given data.

Configures the SHA to compute new running hash as AHB master and returns immediately.
SHA AHB Master mode supports only aligned input address and can be called only once per
continuous block of data. Every call to this function must be preceded with SHA_Init() and
finished with _SHA_Finish(). Once callback function is invoked by SHA isr, it should set

a flag for the main application to finalize the hashing (padding) and to read out the final digest by calling SHA_Finish().

> **Parameters**
>
> - base – SHA peripheral base address
> - ctx – Specifies callback. Last incomplete 512-bit block of the input is copied into clear buffer for padding.
> - input – 32-bit word aligned pointer to Input data.
> - inputSize – Size of input data in bytes (must be word aligned)

> **Returns**
> Status of the hash update operation.

void SHA_ClkInit(SHA_Type *base)

Start SHA clock.

Start SHA clock

> **Parameters**
>
> - base – SHA peripheral base address

void SHA_ClkDeinit(SHA_Type *base)

Stop SHA clock.

Stop SHA clock

> **Parameters**
>
> - base – SHA peripheral base address

2.44   SPI: Serial Peripheral Interface Driver

2.45   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 transfered bytes.

This function gets the master DMA transfered 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 transfered bytes.

This function gets the slave DMA transfered 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.

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

SPI_Type *base

SPI base address

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

uint16_t txDummy

The dummy data for TX.

uint32_t lastword

The last word for master TX.

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

size_t txRemainingBytes

>   lastwordBytes + txRemainingBytes is number of data to be send [in bytes]

uint8_t *rxNextData

>   The pointer of next time rx data

size_t rxRemainingBytes

>   Number of data to be received [in bytes]

bool isSlave

>   SPI work in slave mode.

2.46 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 maxinun 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 maxinun 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

2.47 SPIFI: SPIFI flash interface driver

void SPIFI_TransferTxCreateHandleDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *spifi_dma_callback_t* callback, void *userData, *dma_handle_t* *dmaHandle)

Initializes the SPIFI handle for send which is used in transactional functions and set the callback.

Parameters

- base – SPIFI peripheral base address
- handle – Pointer to spifi_dma_handle_t structure
- callback – SPIFI callback, NULL means no callback.
- userData – User callback function data.
- dmaHandle – User requested DMA handle for DMA transfer

void SPIFI_TransferRxCreateHandleDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *spifi_dma_callback_t* callback, void *userData, *dma_handle_t* *dmaHandle)

Initializes the SPIFI handle for receive which is used in transactional functions and set the callback.

Parameters

- base – SPIFI peripheral base address
- handle – Pointer to spifi_dma_handle_t structure

- callback – SPIFI callback, NULL means no callback.

- userData – User callback function data.

- dmaHandle – User requested DMA handle for DMA transfer

status_t SPIFI_TransferSendDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *spifi_transfer_t* *xfer)

Transfers SPIFI data using an DMA non-blocking method.

This function writes data to the SPIFI transmit FIFO. This function is non-blocking.

Parameters

- base – Pointer to QuadSPI Type.

- handle – Pointer to spifi_dma_handle_t structure

- xfer – SPIFI transfer structure.

status_t SPIFI_TransferReceiveDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *spifi_transfer_t* *xfer)

Receives data using an DMA non-blocking method.

This function receive data from the SPIFI receive buffer/FIFO. This function is non-blocking.

Parameters

- base – Pointer to QuadSPI Type.

- handle – Pointer to spifi_dma_handle_t structure

- xfer – SPIFI transfer structure.

void SPIFI_TransferAbortSendDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle)

Aborts the sent data using DMA.

This function aborts the sent data using DMA.

Parameters

- base – SPIFI peripheral base address.

- handle – Pointer to spifi_dma_handle_t structure

void SPIFI_TransferAbortReceiveDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle)

Aborts the receive data using DMA.

This function abort receive data which using DMA.

Parameters

- base – SPIFI peripheral base address.

- handle – Pointer to spifi_dma_handle_t structure

status_t SPIFI_TransferGetSendCountDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, size_t *count)

Gets the transferred counts of send.

Parameters

- base – Pointer to QuadSPI Type.

- handle – Pointer to spifi_dma_handle_t structure.

- count – Bytes sent.

Return values

- kStatus_Success – Succeed get the transfer count.

> • kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

status_t SPIFI_TransferGetReceiveCountDMA(SPIFI_Type *base, *spifi_dma_handle_t* *handle, size_t *count)

> Gets the status of the receive transfer.

> > **Parameters**

> > > • base – Pointer to QuadSPI Type.

> > > • handle – Pointer to spifi_dma_handle_t structure

> > > • count – Bytes received.

> > **Return values**

> > > • kStatus_Success – Succeed get the transfer count.

> > > • kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

uint32_t SPIFI_GetInstance(SPIFI_Type *base)

> Get the SPIFI instance from peripheral base address.

> > **Parameters**

> > > • base – SPIFI peripheral base address.

> > **Returns**

> > > SPIFI instance.

void SPIFI_Init(SPIFI_Type *base, const *spifi_config_t* *config)

> Initializes the SPIFI with the user configuration structure.

> This function configures the SPIFI module with the user-defined configuration.

> > **Parameters**

> > > • base – SPIFI peripheral base address.

> > > • config – The pointer to the configuration structure.

void SPIFI_GetDefaultConfig(*spifi_config_t* *config)

> Get SPIFI default configure settings.

> > **Parameters**

> > > • config – SPIFI config structure pointer.

void SPIFI_Deinit(SPIFI_Type *base)

> Deinitializes the SPIFI regions.

> > **Parameters**

> > > • base – SPIFI peripheral base address.

void SPIFI_SetCommand(SPIFI_Type *base, *spifi_command_t* *cmd)

> Set SPIFI flash command.

> > **Parameters**

> > > • base – SPIFI peripheral base address.

> > > • cmd – SPIFI command structure pointer.

static inline void SPIFI_SetCommandAddress(SPIFI_Type *base, uint32_t addr)

> Set SPIFI command address.

> > **Parameters**

- base – SPIFI peripheral base address.

- addr – Address value for the command.

static inline void SPIFI_SetIntermediateData(SPIFI_Type *base, uint32_t val)

Set SPIFI intermediate data.

Before writing a command wihch needs specific intermediate value, users shall call this function to write it. The main use of this function for current serial flash is to select no-opcode mode and cancelling this mode. As dummy cycle do not care about the value, no need to call this function.

Parameters

- base – SPIFI peripheral base address.

- val – Intermediate data.

static inline void SPIFI_SetCacheLimit(SPIFI_Type *base, uint32_t val)

Set SPIFI Cache limit value.

SPIFI includes caching of prevously-accessed data to improve performance. Software can write an address to this function, to prevent such caching at and above the address.

Parameters

- base – SPIFI peripheral base address.

- val – Zero-based upper limit of cacheable memory.

static inline void SPIFI_ResetCommand(SPIFI_Type *base)

Reset the command field of SPIFI.

This function is used to abort the current command or memory mode.

Parameters

- base – SPIFI peripheral base address.

void SPIFI_SetMemoryCommand(SPIFI_Type *base, *spifi_command_t* *cmd)

Set SPIFI flash AHB read command.

Call this function means SPIFI enters to memory mode, while users need to use command, a SPIFI_ResetCommand shall be called.

Parameters

- base – SPIFI peripheral base address.

- cmd – SPIFI command structure pointer.

static inline void SPIFI_EnableInterrupt(SPIFI_Type *base, uint32_t mask)

Enable SPIFI interrupt.

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

- base – SPIFI peripheral base address.

- mask – SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: spifi_interrupt_enable_t

static inline void SPIFI_DisableInterrupt(SPIFI_Type *base, uint32_t mask)

Disable SPIFI interrupt.

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

- base – SPIFI peripheral base address.

- mask – SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: spifi_interrupt_enable_t

static inline uint32_t SPIFI_GetStatusFlag(SPIFI_Type *base)

Get the status of all interrupt flags for SPIFI.

Parameters

- base – SPIFI peripheral base address.

Returns

SPIFI flag status

FSL_SPIFI_DMA_DRIVER_VERSION

SPIFI DMA driver version 2.0.3.

FSL_SPIFI_DRIVER_VERSION

SPIFI driver version 2.0.3.

Status structure of SPIFI.

Values:

enumerator kStatus_SPIFI_Idle

SPIFI is in idle state

enumerator kStatus_SPIFI_Busy

SPIFI is busy

enumerator kStatus_SPIFI_Error

Error occurred during SPIFI transfer

enum _spifi_interrupt_enable

SPIFI interrupt source.

Values:

enumerator kSPIFI_CommandFinishInterruptEnable

Interrupt while command finished

enum _spifi_spi_mode

SPIFI SPI mode select.

Values:

enumerator kSPIFI_SPISckLow

SCK low after last bit of command, keeps low while CS high

enumerator kSPIFI_SPISckHigh

SCK high after last bit of command and while CS high

enum _spifi_dual_mode

SPIFI dual mode select.

Values:

enumerator kSPIFI_QuadMode

SPIFI uses IO3:0

enumerator kSPIFI_DualMode

SPIFI uses IO1:0

enum __spifi_data_direction
> SPIFI data direction.

> *Values:*

> enumerator kSPIFI_DataInput
>> Data input from serial flash.

> enumerator kSPIFI_DataOutput
>> Data output to serial flash.

enum __spifi_command_format
> SPIFI command opcode format.

> *Values:*

> enumerator kSPIFI_CommandAllSerial
>> All fields of command are serial.

> enumerator kSPIFI_CommandDataQuad
>> Only data field is dual/quad, others are serial.

> enumerator kSPIFI_CommandOpcodeSerial
>> Only opcode field is serial, others are quad/dual.

> enumerator kSPIFI_CommandAllQuad
>> All fields of command are dual/quad mode.

enum __spifi_command_type
> SPIFI command type.

> *Values:*

> enumerator kSPIFI_CommandOpcodeOnly
>> Command only have opcode, no address field

> enumerator kSPIFI_CommandOpcodeAddrOneByte
>> Command have opcode and also one byte address field

> enumerator kSPIFI_CommandOpcodeAddrTwoBytes
>> Command have opcode and also two bytes address field

> enumerator kSPIFI_CommandOpcodeAddrThreeBytes
>> Command have opcode and also three bytes address field.

> enumerator kSPIFI_CommandOpcodeAddrFourBytes
>> Command have opcode and also four bytes address field

> enumerator kSPIFI_CommandNoOpcodeAddrThreeBytes
>> Command have no opcode and three bytes address field

> enumerator kSPIFI_CommandNoOpcodeAddrFourBytes
>> Command have no opcode and four bytes address field

> SPIFI status flags.

> *Values:*

> enumerator kSPIFI_MemoryCommandWriteFinished
>> Memory command write finished

> enumerator kSPIFI_CommandWriteFinished
>> Command write finished

2.47. SPIFI: SPIFI flash interface driver   **437**

enumerator kSPIFI_InterruptRequest
> CMD flag from 1 to 0, means command execute finished

typedef struct _*spifi_dma_handle* spifi_dma_handle_t

typedef void (*spifi_dma_callback_t)(SPIFI_Type *base, *spifi_dma_handle_t* *handle, *status_t* status, void *userData)
> SPIFI DMA transfer callback function for finish and error.

typedef enum _*spifi_interrupt_enable* spifi_interrupt_enable_t
> SPIFI interrupt source.

typedef enum _*spifi_spi_mode* spifi_spi_mode_t
> SPIFI SPI mode select.

typedef enum _*spifi_dual_mode* spifi_dual_mode_t
> SPIFI dual mode select.

typedef enum _*spifi_data_direction* spifi_data_direction_t
> SPIFI data direction.

typedef enum _*spifi_command_format* spifi_command_format_t
> SPIFI command opcode format.

typedef enum _*spifi_command_type* spifi_command_type_t
> SPIFI command type.

typedef struct _*spifi_command* spifi_command_t
> SPIFI command structure.

typedef struct _*spifi_config* spifi_config_t
> SPIFI region configuration structure.

typedef struct _*spifi_transfer* spifi_transfer_t
> Transfer structure for SPIFI.

static inline void SPIFI_EnableDMA(SPIFI_Type *base, bool enable)
> Enable or disable DMA request for SPIFI.

> **Parameters**
> > - base – SPIFI peripheral base address.
> > - enable – True means enable DMA and false means disable DMA.

static inline uint32_t SPIFI_GetDataRegisterAddress(SPIFI_Type *base)
> Gets the SPIFI data register address.

> This API is used to provide a transfer address for the SPIFI DMA transfer configuration.

> **Parameters**
> > - base – SPIFI base pointer

> **Returns**
> > data register address

static inline void SPIFI_WriteData(SPIFI_Type *base, uint32_t data)
> Write a word data in address of SPIFI.

> Users can write a page or at least a word data into SPIFI address.

> **Parameters**
> > - base – SPIFI peripheral base address.
> > - data – Data need be write.

static inline void SPIFI_WriteDataByte(SPIFI_Type *base, uint8_t data)

> Write a byte data in address of SPIFI.

> Users can write a byte data into SPIFI address.

>> **Parameters**

>>> • base – SPIFI peripheral base address.

>>> • data – Data need be write.

void SPIFI_WriteDataHalfword(SPIFI_Type *base, uint16_t data)

> Write a halfword data in address of SPIFI.

> Users can write a halfword data into SPIFI address.

>> **Parameters**

>>> • base – SPIFI peripheral base address.

>>> • data – Data need be write.

static inline uint32_t SPIFI_ReadData(SPIFI_Type *base)

> Read data from serial flash.

> Users should notice before call this function, the data length field in command register shall larger than 4, otherwise a hardfault will happen.

>> **Parameters**

>>> • base – SPIFI peripheral base address.

>> **Returns**

>>> Data input from flash.

static inline uint8_t SPIFI_ReadDataByte(SPIFI_Type *base)

> Read a byte data from serial flash.

>> **Parameters**

>>> • base – SPIFI peripheral base address.

>> **Returns**

>>> Data input from flash.

uint16_t SPIFI_ReadDataHalfword(SPIFI_Type *base)

> Read a halfword data from serial flash.

>> **Parameters**

>>> • base – SPIFI peripheral base address.

>> **Returns**

>>> Data input from flash.

struct _spifi_dma_handle

> *#include <fsl_spifi_dma.h>* SPIFI DMA transfer handle, users should not touch the content of the handle.

> **Public Members**

> *dma_handle_t* *dmaHandle

>> DMA handler for SPIFI send

> size_t transferSize

>> Bytes need to transfer.

uint32_t state

    Internal state for SPIFI DMA transfer

spifi_dma_callback_t callback

    Callback for users while transfer finish or error occurred

void *userData

    User callback parameter

struct __spifi__command

    #include <fsl_spifi.h> SPIFI command structure.

Public Members

uint16_t dataLen

    How many data bytes are needed in this command.

bool isPollMode

    For command need to read data from serial flash

spifi_data_direction_t direction

    Data direction of this command.

uint8_t intermediateBytes

    How many intermediate bytes needed

spifi_command_format_t format

    Command format

spifi_command_type_t type

    Command type

uint8_t opcode

    Command opcode value

struct __spifi__config

    #include <fsl_spifi.h> SPIFI region configuration structure.

Public Members

uint16_t timeout

    SPI transfer timeout, the unit is SCK cycles

uint8_t csHighTime

    CS high time cycles

bool disablePrefetch

    True means SPIFI will not attempt a speculative prefetch.

bool disableCachePrefech

    Disable prefetch of cache line

bool isFeedbackClock

    Is data sample uses feedback clock.

spifi_spi_mode_t spiMode

    SPIFI spi mode select

bool isReadFullClockCycle

    If enable read full clock cycle.

spifi_dual_mode_t dualMode
    SPIFI dual mode, dual or quad.

struct __spifi_transfer
    #include <fsl_spifi.h> Transfer structure for SPIFI.

Public Members

uint8_t *data
    Pointer to data to transmit

size_t dataSize
    Bytes to be transmit

2.48 SPIFI DMA Driver

2.49 SPIFI Driver

2.50 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.51 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

2.52 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 manully enabled by user, by calling RTC_Init and setting SYSCON_RTCOSCCTRL_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 receices 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 funciton, 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 cuntion, 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 ___unnamed36___

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.

2.53 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

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

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

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.

2.54   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 0xFFFFFF (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 0xFFFFFF 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 Connectivity

3.1.1 lwIP

This is the NXP fork of the lwIP networking stack.

- For details about changes and additions made by NXP, see CHANGELOG.
- For details about the NXP porting layer, see *The NXP lwIP Port*.
- For usage and API of lwIP, use official documentation at http://www.nongnu.org/lwip/.

The NXP lwIP Port

Below is description of possible settings of the port layer and an overview of a few helper functions.

The best place for redefinition of any mentioned macro is lwipopts.h.

The declaration of every mentioned function is in ethernetif.h. Please check the doxygen comments of those functions before.

Link state   Physical link state (up/down) and its speed and duplex must be read out from PHY over MDIO bus. Especially link information is useful for lwIP stack so it can for example send DHCP discovery immediately when a link becomes up.

To simplify this port layer offers a function ethernetif_probe_link() which reads those data from PHY and forwards them into lwIP stack.

In almost all examples this function is called every ETH_LINK_POLLING_INTERVAL_MS (1500ms) by a function probe_link_cyclic().

By setting ETH_LINK_POLLING_INTERVAL_MS to 0 polling will be disabled. On FreeRTOS, probe_link_cyclic() will be then called on an interrupt generated by PHY. GPIO port and pin for the interrupt line must be set in the ethernetifConfig struct passed to ethernetif_init(). On bare metal interrupts are not supported right now.

Rx task   To improve the reaction time of the app, reception of packets is done in a dedicated task. The rx task stack size can be set by ETH_RX_TASK_STACK_SIZE macro, its priority by ETH_RX_TASK_PRIO.

If you want to save memory you can set reception to be done in an interrupt by setting ETH_DO_RX_IN_SEPARATE_TASK macro to 0.

Disabling Rx interrupt when out of buffers   If ETH_DISABLE_RX_INT_WHEN_OUT_OF_BUFFERS is set to 1, then when the port gets out of Rx buffers, Rx enet interrupt will be disabled for a particular controller. Everytime Rx buffer is freed, Rx interrupt will be enabled.

This prevents your app from never getting out of Rx interrupt when the network is flooded with traffic.

ETH_DISABLE_RX_INT_WHEN_OUT_OF_BUFFERS is by default turned on, on FreeRTOS and off on bare metal.

Limit the number of packets read out from the driver at once on bare metal.   You may define macro ETH_MAX_RX_PKTS_AT_ONCE to limit the number of received packets read out from the driver at once.

In case of heavy Rx traffic, lowering this number improves the realtime behaviour of an app. Increasing improves Rx throughput.

Setting it to value < 1 or not defining means "no limit".

Helper functions   If your application needs to wait for the link to become up you can use one of the following functions:

- ethernetif_wait_linkup()- Blocks until the link on the passed netif is not up.

- ethernetif_wait_linkup_array() - Blocks until the link on at least one netif from the passed list of netifs becomes up.

If your app needs to wait for the IPv4 address on a particular netif to become different than "ANY" address (255.255.255.255) function ethernetif_wait_ipv4_valid() does this.

3.2 File System

3.2.1 FatFs

MCUXpresso SDK : mcuxsdk-middleware-fatfs

Overview   This repository is for FatFs middleware delivery and it contains the components officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository (mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

Documentation   Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit FatFs - Documentation to review details on the contents in this sub-repo.

Setup   Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution   Contributions are not currently accepted. Guidelines to contribute will be posted in the future.

Repo Specific Content This is MCUXpresso SDK fork of FatFs (FAT file system created by ChaN). Official documentation is available at http://elm-chan.org/fsw/ff/

MCUXpresso version is extending original content by following hardware specific porting layers:

- mmc_disk
- nand_disk
- ram_disk
- sd_disk
- sdspi_disk
- usb_disk

Changelog FatFs

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog

[R0.15_rev0]

- Upgraded to version 0.15
- Applied patches from http://elm-chan.org/fsw/ff/patches.html

[R0.14b_rev1]

- Applied patches from http://elm-chan.org/fsw/ff/patches.html

[R0.14b_rev0]

- Upgraded to version 0.14b

[R0.14a_rev0]

- Upgraded to version 0.14a
- Applied patch ff14a_p1.diff and ff14a_p2.diff

[R0.14_rev0]

- Upgraded to version 0.14
- Applied patch ff14_p1.diff and ff14_p2.diff

[R0.13c_rev0]

- Upgraded to version 0.13c
- Applied patches ff_13c_p1.diff,ff_13c_p2.diff, ff_13c_p3.diff and ff_13c_p4.diff.

[R0.13b_rev0]

- Upgraded to version 0.13b

[R0.13a_rev0]

- Upgraded to version 0.13a. Added patch ff_13a_p1.diff.

[R0.12c_rev1]

- Add NAND disk support.

[R0.12c_rev0]

- Upgraded to version 0.12c and applied patches ff_12c_p1.diff and ff_12c_p2.diff.

[R0.12b_rev0]

- Upgraded to version 0.12b.

[R0.11a]

- Added glue functions for low-level drivers (SDHC, SDSPI, RAM, MMC). Modified diskio.c.
- Added RTOS wrappers to make FatFs thread safe. Modified syscall.c.
- Renamed ffconf.h to ffconf_template.h. Each application should contain its own ffconf.h.
- Included ffconf.h into diskio.c to enable the selection of physical disk from ffconf.h by macro definition.
- Conditional compilation of physical disk interfaces in diskio.c.

3.3 Motor Control

3.3.1 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 mod-

ules 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 spplications   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 FM-STR_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 FreeMAS-TER 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 FM-STR_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 execu-
tion 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 FreeMAS-
TER 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 FM-
STR_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, FM-*
STR_CanIsr, or other interrupt service routine, while the protocol decoding and execution is han-
dled 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 FM-
STR_CanIsr functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FM-*
STR_Poll function is called at least once per *N* character time periods. *N* is the length of the
FreeMASTER FIFO queue (*FMSTR_COMM_RQUEUE_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-langiage 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 n bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size n bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size n bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(m,n)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(m,n)	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)
FMSTR_TSA_USERTYPE(*name*)	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 *fmstr:* 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 */

/* 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);
```

3.3. Motor Control   **497**

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
FM-STR_ADDR	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.
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.	
FM-STR_SIZE	Data type used to hold the memory block size.
It is required that this type is unsigned and at least 16 bits wide integer.	
FM-STR_BOOL	Data type used as a general boolean type.
This type is used only in zero/non-zero conditions in the driver code.	
FM-STR_APPCM	Data type used to hold the Application Command code.
Generally, this is an unsigned 8-bit value.	
FM-STR_APPCM	Data type used to create the Application Command data buffer.
Generally, this is an unsigned 8-bit value.	
FM-STR_APPCM	Data type used to hold the Application Command result code.
Generally, this is an unsigned 8-bit value.	

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.

FM-STR_TSA_TIl	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 FM-STR_SIZE.	
FM-STR_TSA_TS.	Data type used to hold a memory block size, as used in the TSA descriptors.
By default, this is defined as FM-STR_SIZE.	

Public Pipes types   The table describes the data types used by the FreeMASTER Pipes API:

FM-STR_HPIPE	Pipe handle that identifies the open-pipe object.
Generally, this is a pointer to a void type.	
FM-STR_PIPE_P(	Integer type required to hold at least 7 bits of data.
Generally, this is an unsigned 8-bit or 16-bit type.	
FM-STR_PIPE_SI	Integer type required to hold at least 16 bits of data.
This is used to store the data buffer sizes.	
FM-STR_PPIPEF(	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.

FMSTR_U8	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.	
FMSTR_U16	Unsigned 16-bit integer.
FMSTR_U32	Unsigned 32-bit integer.
FMSTR_S8	Signed 8-bit integer.
FMSTR_S16	Signed 16-bit integer.
FMSTR_S32	Signed 32-bit integer.
FMSTR_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_FLAGS	Data type forming a union with a structure of flag bit-fields.
FMSTR_SIZE8	Data type holding a general size value, at least 8 bits wide.
FMSTR_INDEX	General for-loop index. Must be signed, at least 16 bits wide.
FMSTR_BCHR	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.	
FMSTR_BPTR	A pointer to the communication buffer (an array of FMSTR_BCHR).

Document references

Links

- This document online: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/middleware/freemaster/doc/index.html

---

- FreeMASTER tool home: www.nxp.com/freemaster

- FreeMASTER community area: community.nxp.com/community/freemaster

- FreeMASTER GitHub code repo: https://github.com/nxp-mcuxpresso/mcux-freemaster

- MCUXpresso SDK home: www.nxp.com/mcuxpresso

- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

- *FreeMASTER Usage Serial Driver Implementation* (document AN4752)

- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document AN4771)

- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document AN4860)

Revision history   This Table summarizes the changes done to this document since the initial release.

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support.
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. added MCAN interface. Folders structure at the installation destination was rearranged.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.
4.3	10/2024	Added description of Network and Segger J-Link RTT interface configuration. Accompanying the MCUXpresso SDK version 24.12.00.
4.4	04/2025	Added Zephyr-specific information. Accompanying the MCUXpresso SDK version 25.06.00.

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

FreeRTOS kernel for MCUXpresso SDK Readme

FreeRTOS kernel for MCUXpresso SDK ChangeLog

FreeRTOS kernel Readme

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

Readme

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme