MCUXpresso SDK Documentation

Release 25.12.00

NXP
Dec 18, 2025

Table of contents

This documentation contains information specific to the lpcxpresso55s28 board.

Chapter 1

LPCXpresso55S28

1.1 Overview

The LPCXpresso55S28 board is an LPCXpresso V3 style board, the latest generation of the original and highly successful LPCXpresso board family. “Arduino UNO” compatible shield connectors are included, with additional expansion ports around the Arduino footprint, along with a PMod/host interface port and Mikroelekronika Click module site. The board features 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 LITE from SEGGER. The board can also be used with an external debug probe such as those from SEGGER and P&E.

The LPC5500 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.

MCU device and part on board is shown below:

- Device: LPC55S28
- PartNumber: LPC55S28JBD100

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.

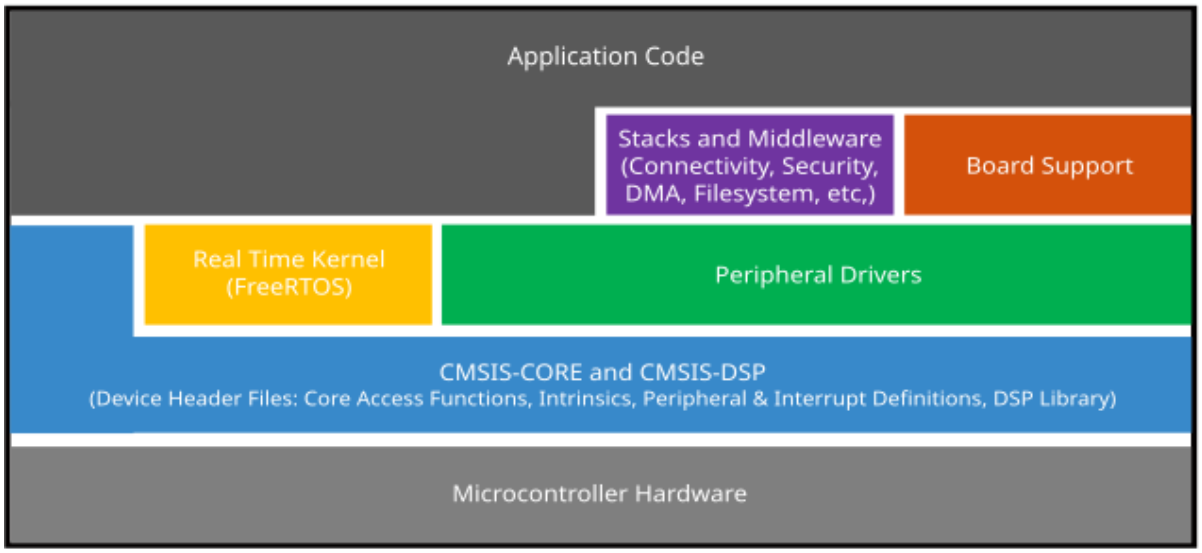
Note: The Repository-Layout SDK package was first introduced in version 25.09.00, but initially only for MCXW23x platforms.

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

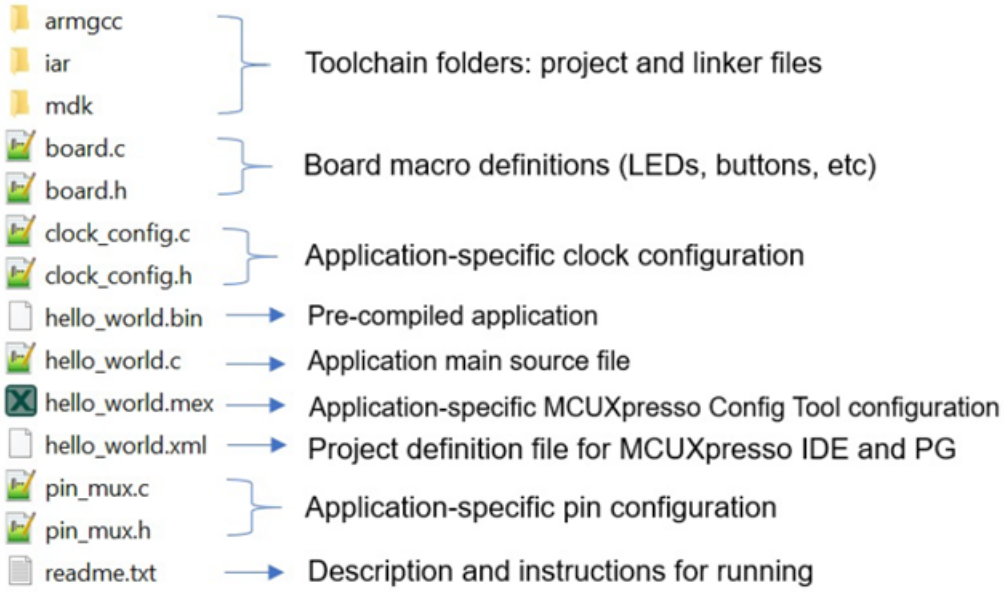
found inside the top-level boards folder and each supported board has its own folder (an MCUXpresso SDK package can support multiple boards). Within each <board_name> folder, there are various subfolders to classify the type of examples it contains. These include (but are not limited to):

- 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.

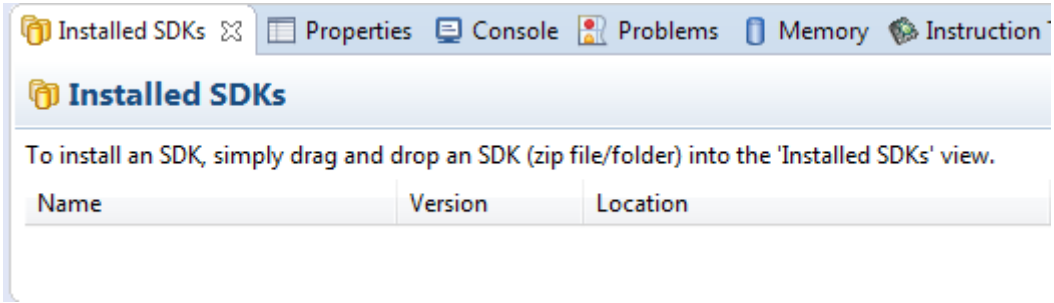
Run a demo using MCUXpresso IDE **Note:** Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

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.

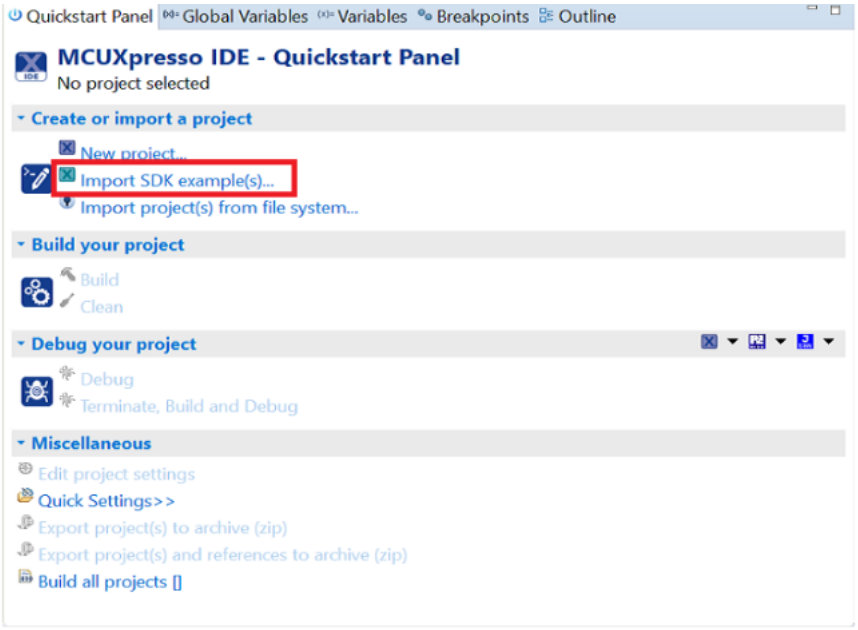
Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace. 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.

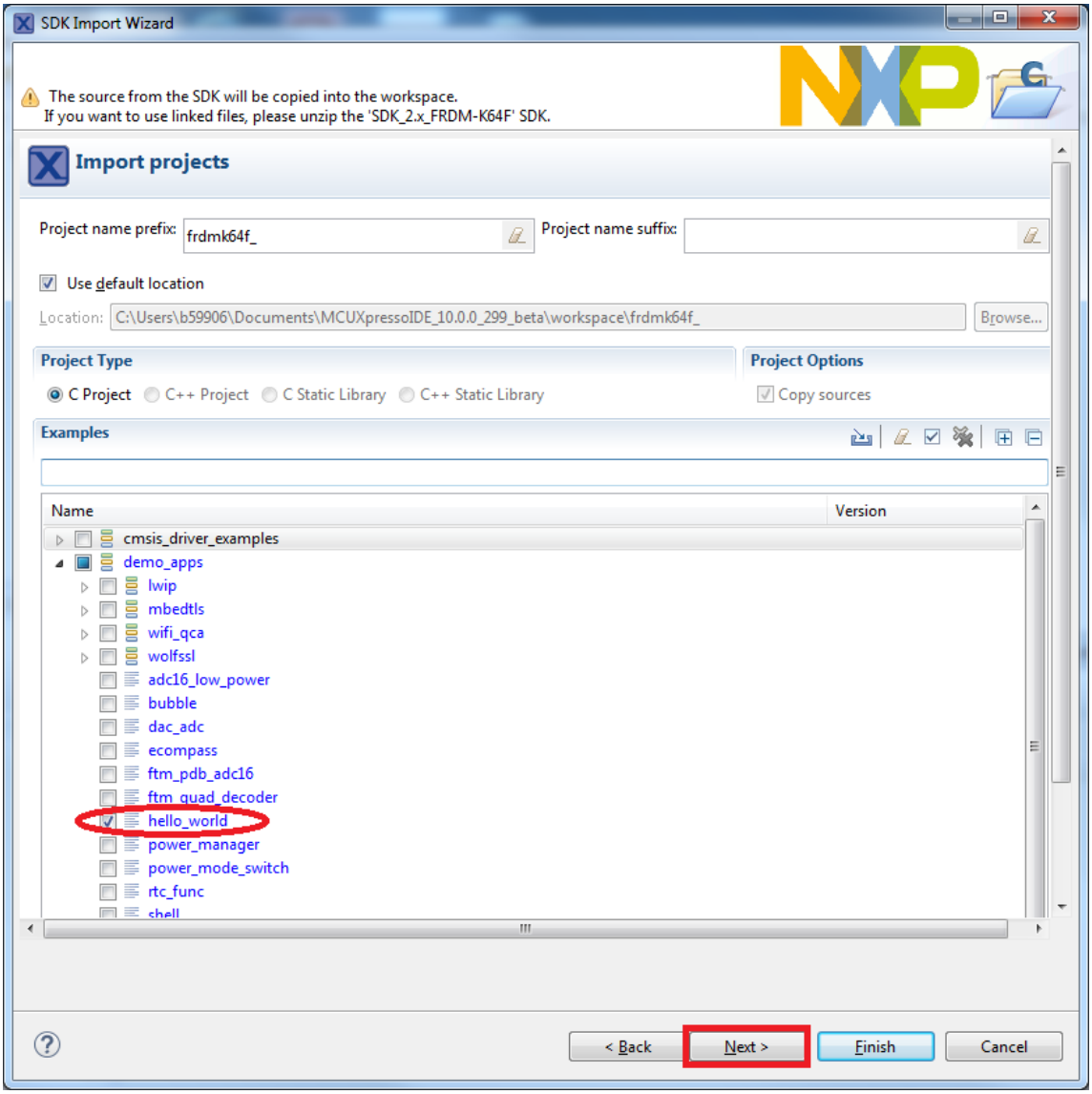
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



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



3. Expand the demo_apps folder and select hello_world.
4. Click **Next.**



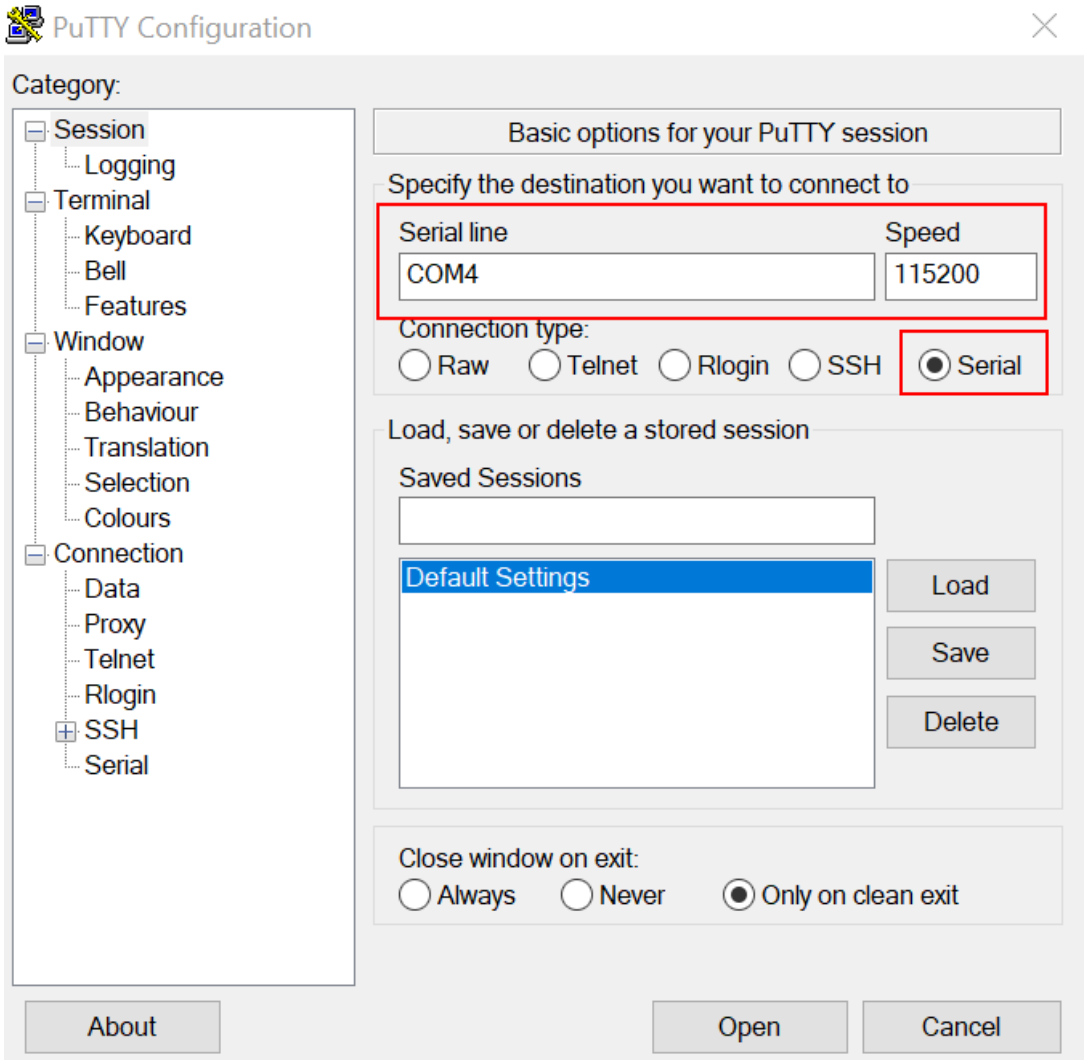
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminalfor demo applications such as adc_basic, adc_burst, adc_dma, and adc_interrupt. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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

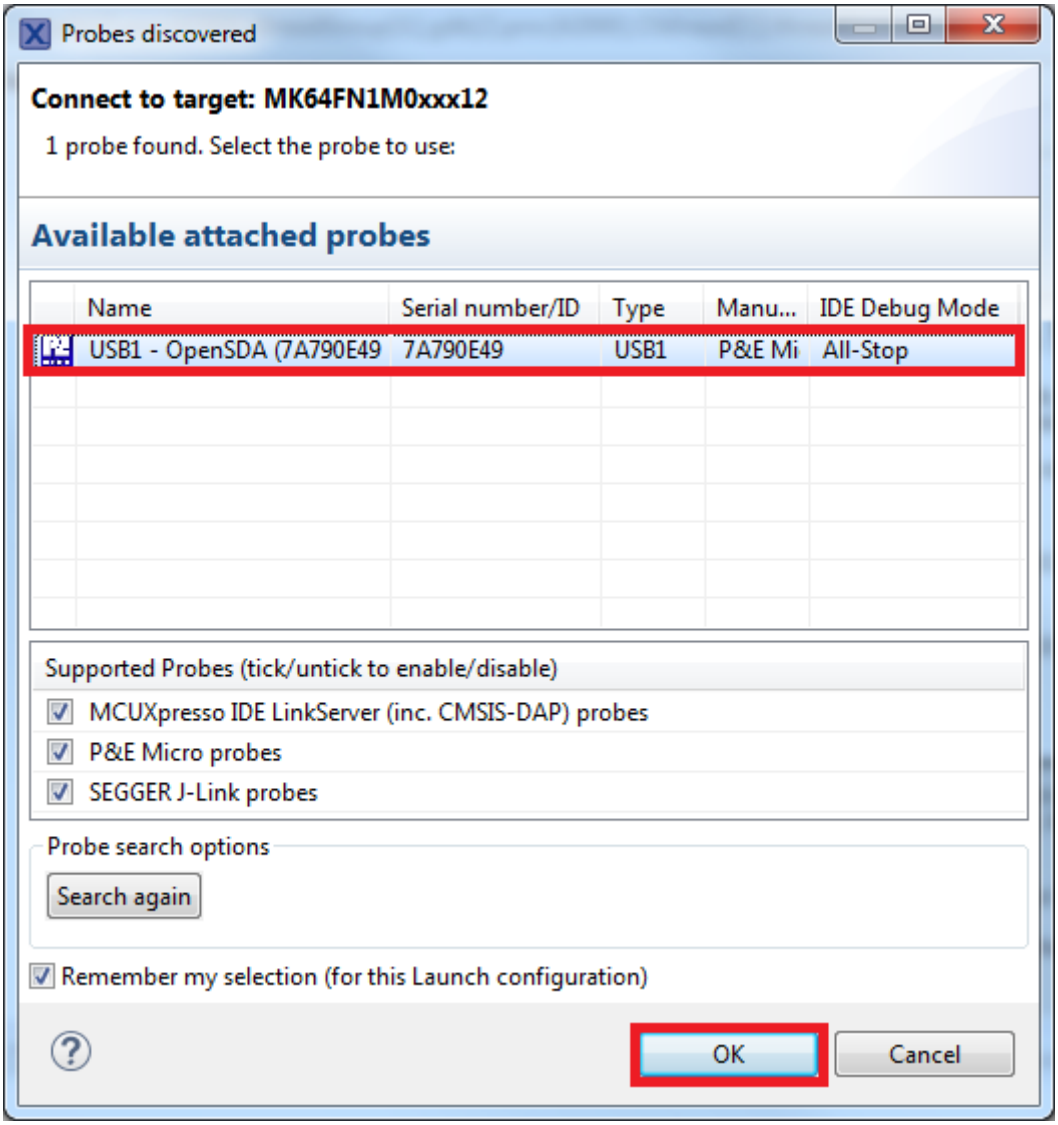
3. 8 data bits



4. 1 stop bit

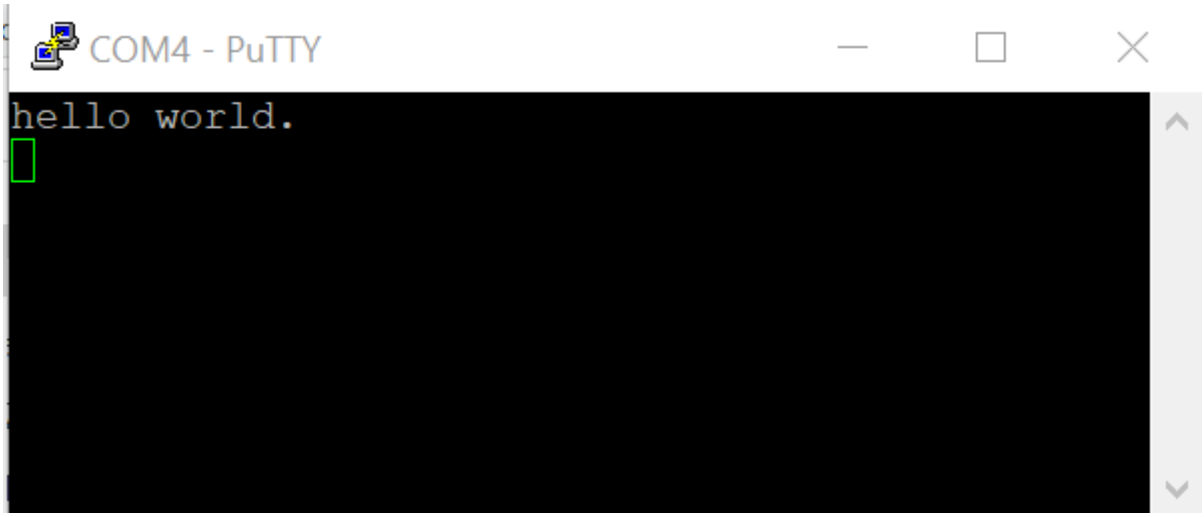
4. On the **Quickstart Panel**, click **Debug** to launch the debug session.

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.)



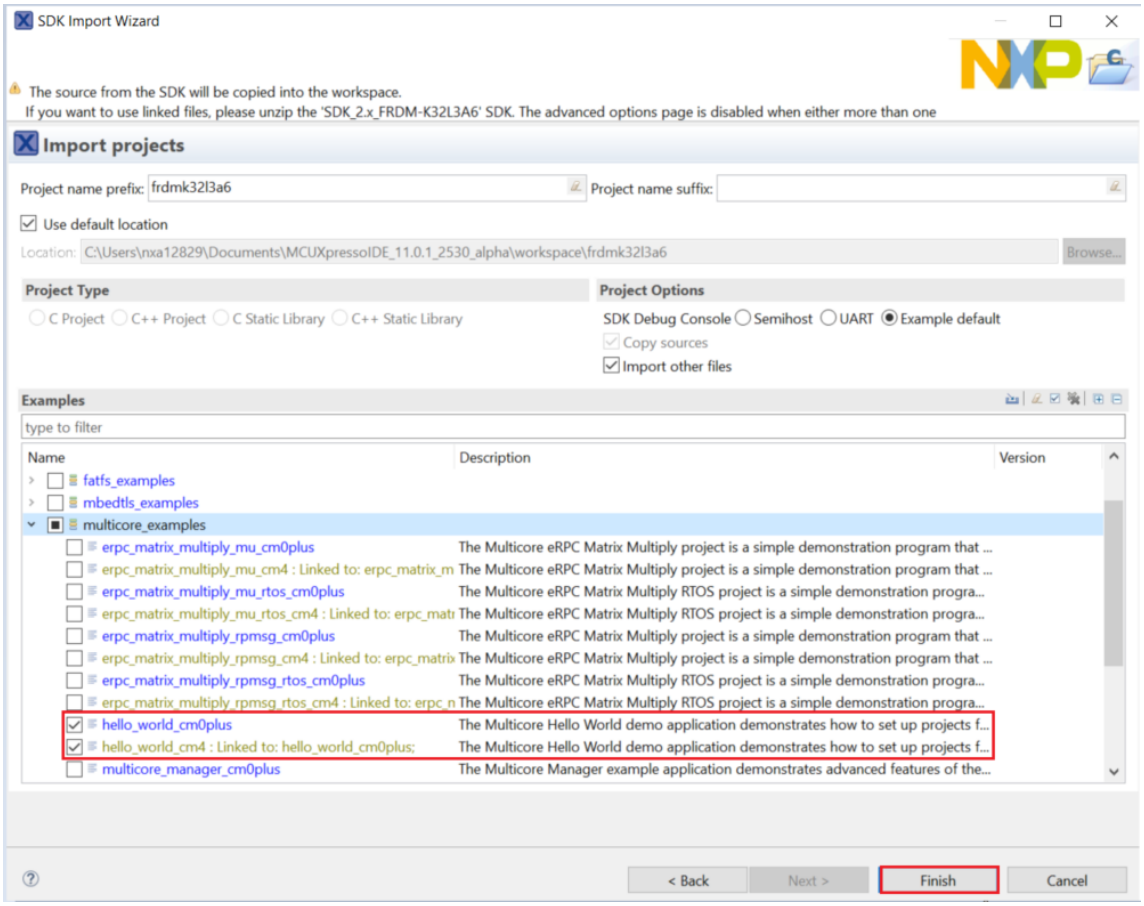
6. The application is downloaded to the target and automatically runs to main().
7. Start the application by clicking **Resume**.

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

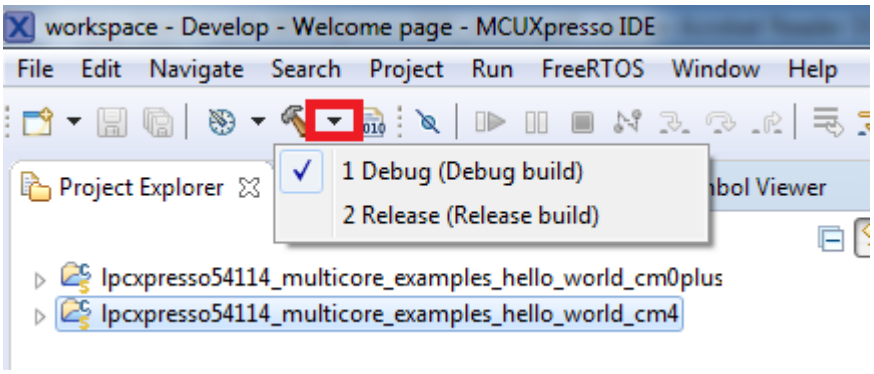


Build a multicore example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello_world 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**.

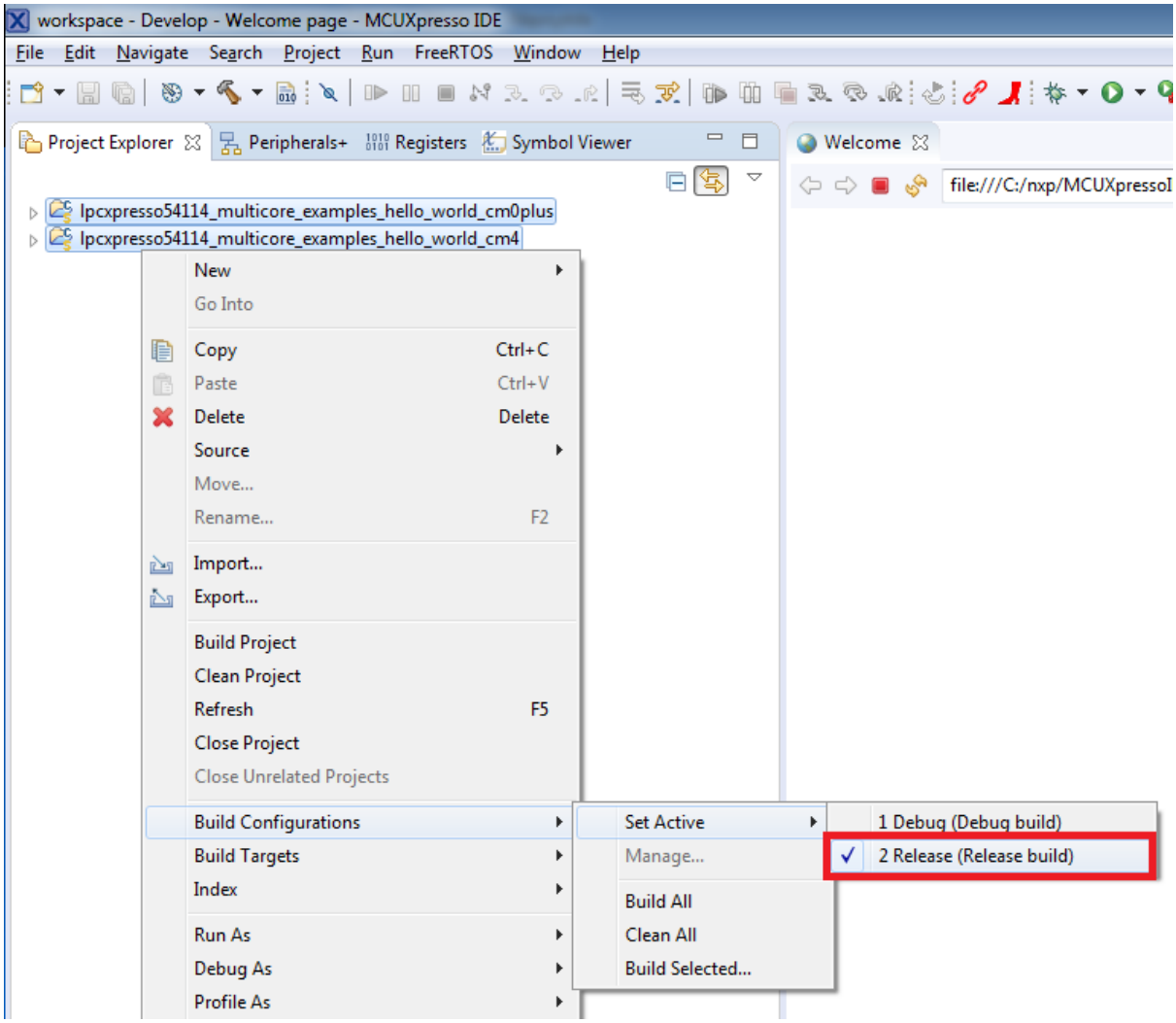


3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the lpcxpresso54114_multicore_examples_hello_world_cm4 project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in the figure. For this example, select **Debug**.

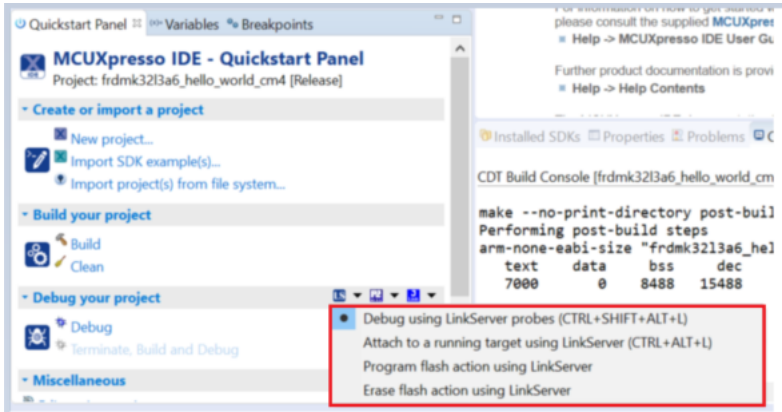


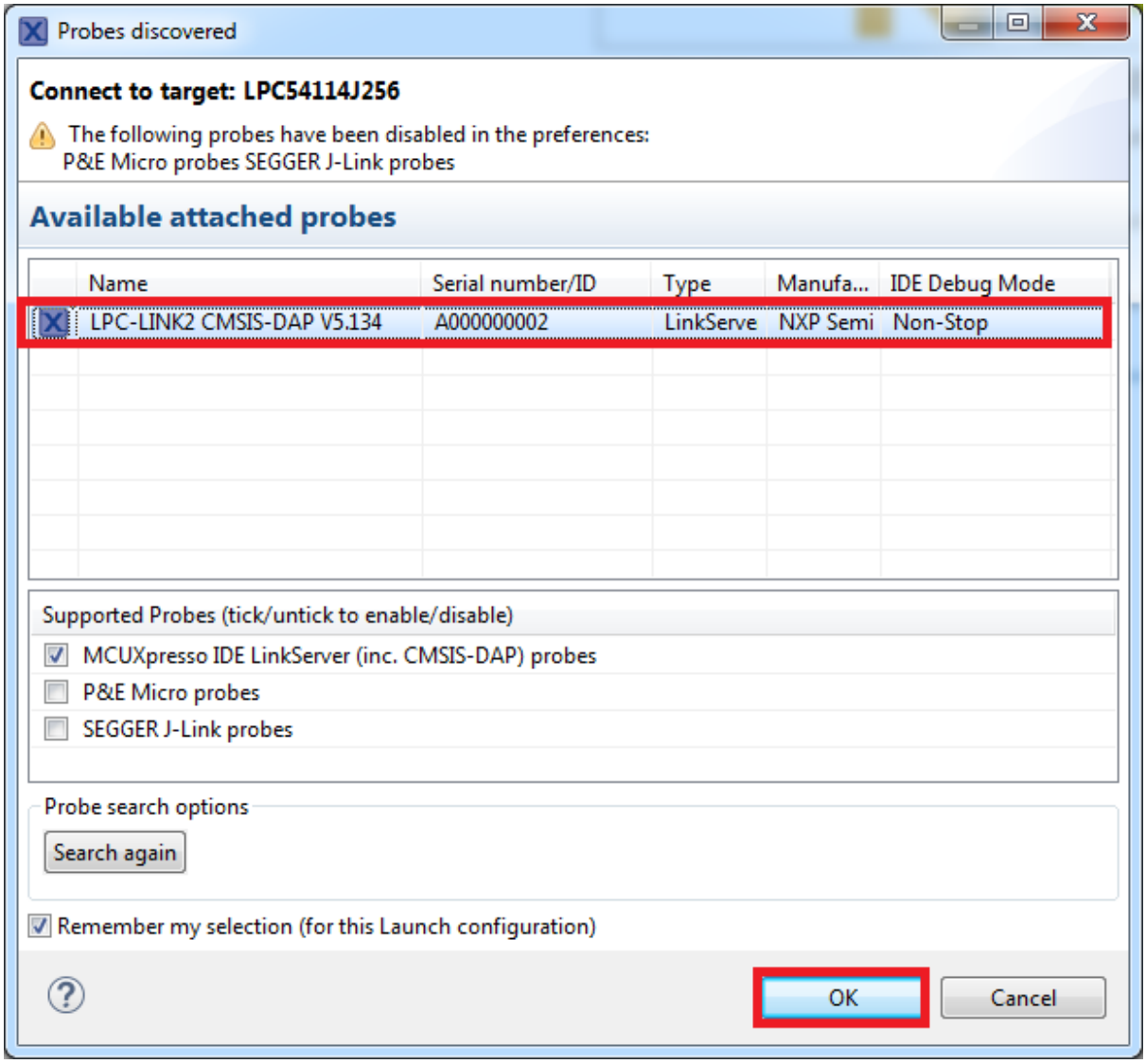
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

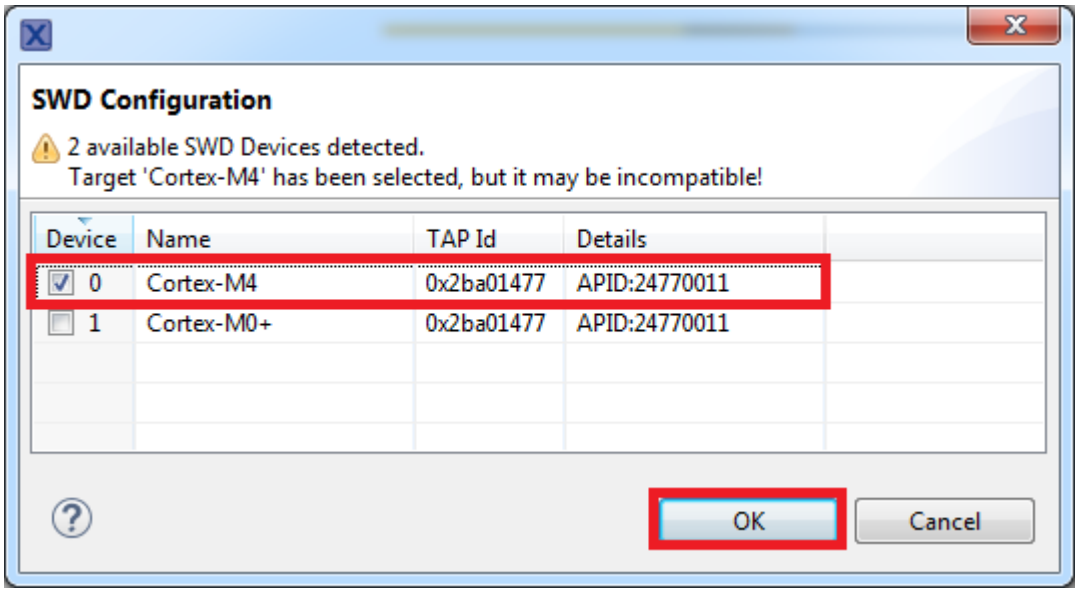
Note: When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **Build 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.

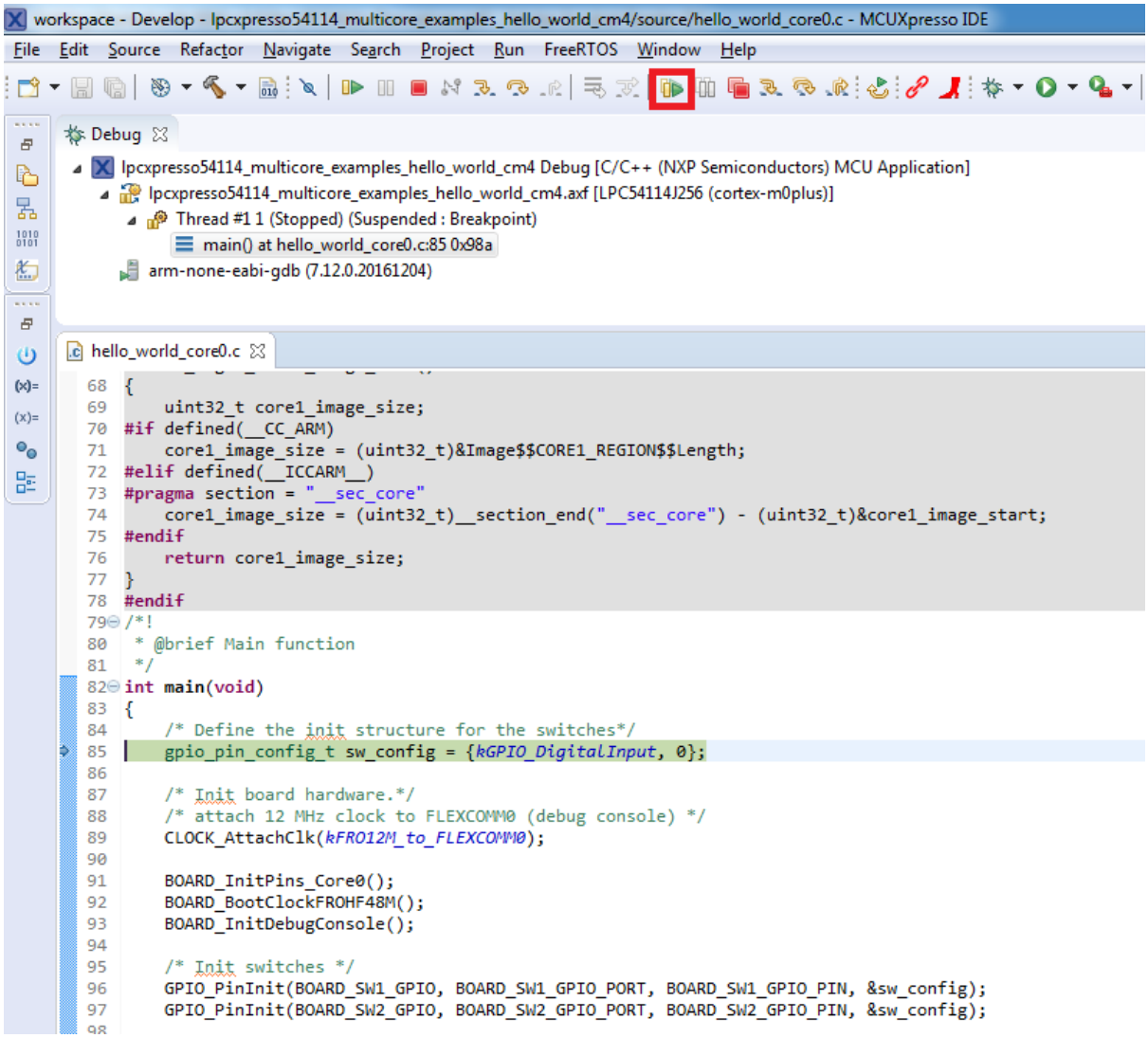


Run a multicore example application The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run an 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.

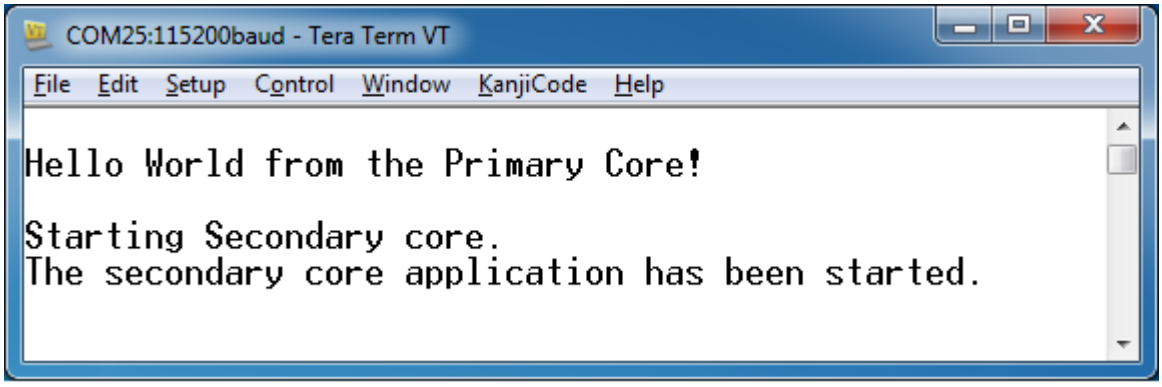



Probes discovered
Connect to target: LPC54114J256
The following probes have been disabled in the preferences:
P&E Micro probes SEGGER J-Link probes
Available attached probes
Name
Serial number/ID
Type
Manufa...
IDE Debug Mode
LPC-LINK2 CMSIS-DAP V5.134
A000000002
LinkServe
NXP Semi
Non-Stop
Supported Probes (tick/untick to enable/disable)
MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
P&E Micro probes
SEGGER J-Link probes
Probe search options
Search again
Remember my selection (for this Launch configuration)
OK
Cancel


SWD Configuration
2 available SWD Devices detected.
Target 'Cortex-M4' has been selected, but it may be incompatible!
Device
Name
TAP Id
Details
0
Cortex-M4
0x2ba01477
APID:24770011
1
Cortex-M0+
0x2ba01477
APID:24770011
OK
Cancel

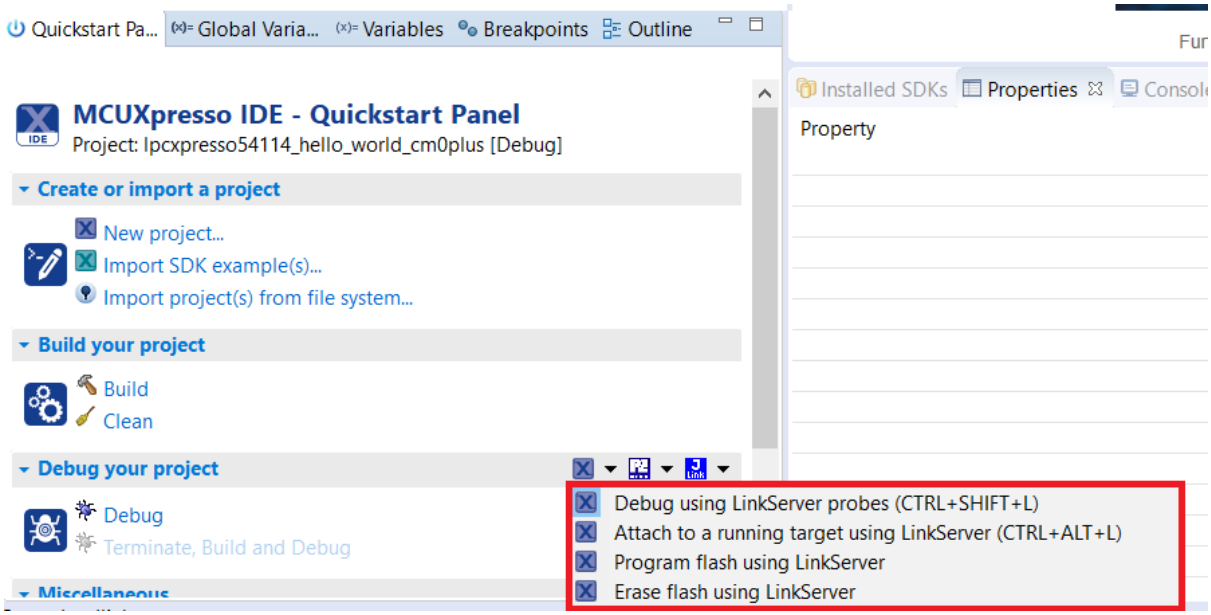


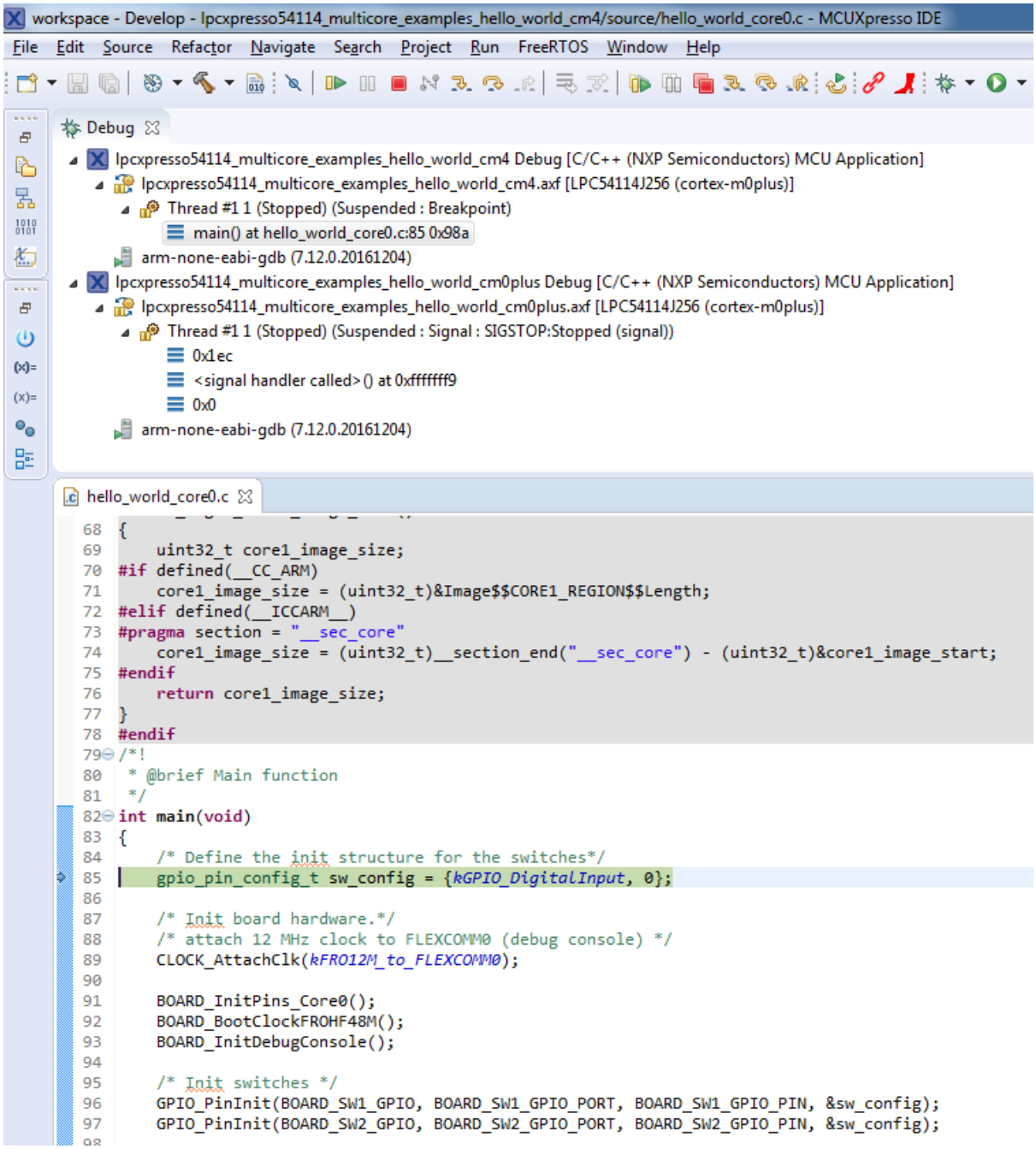
After clicking the “Resume All Debug sessions” button, the hello_world multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



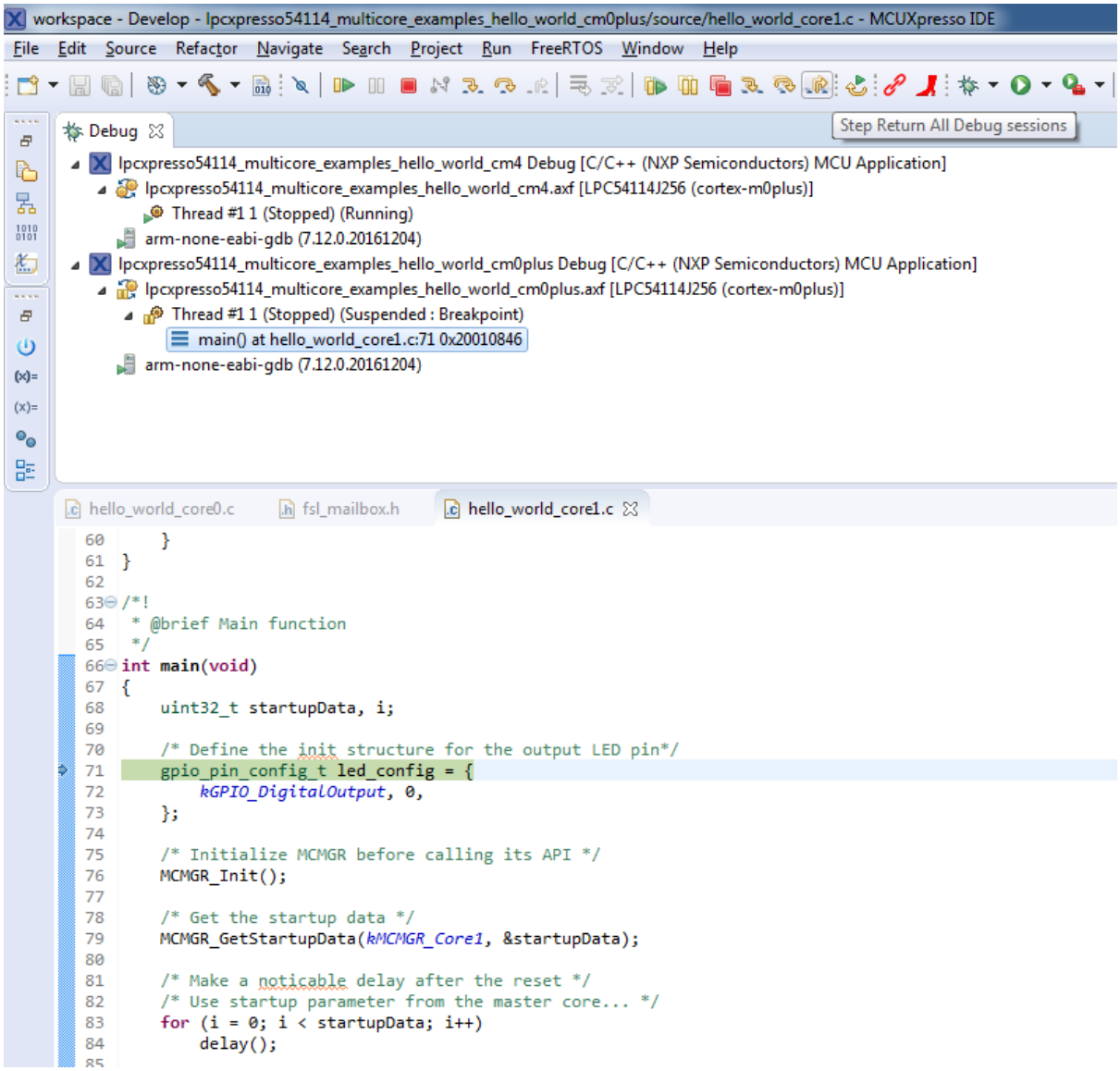
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps 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.

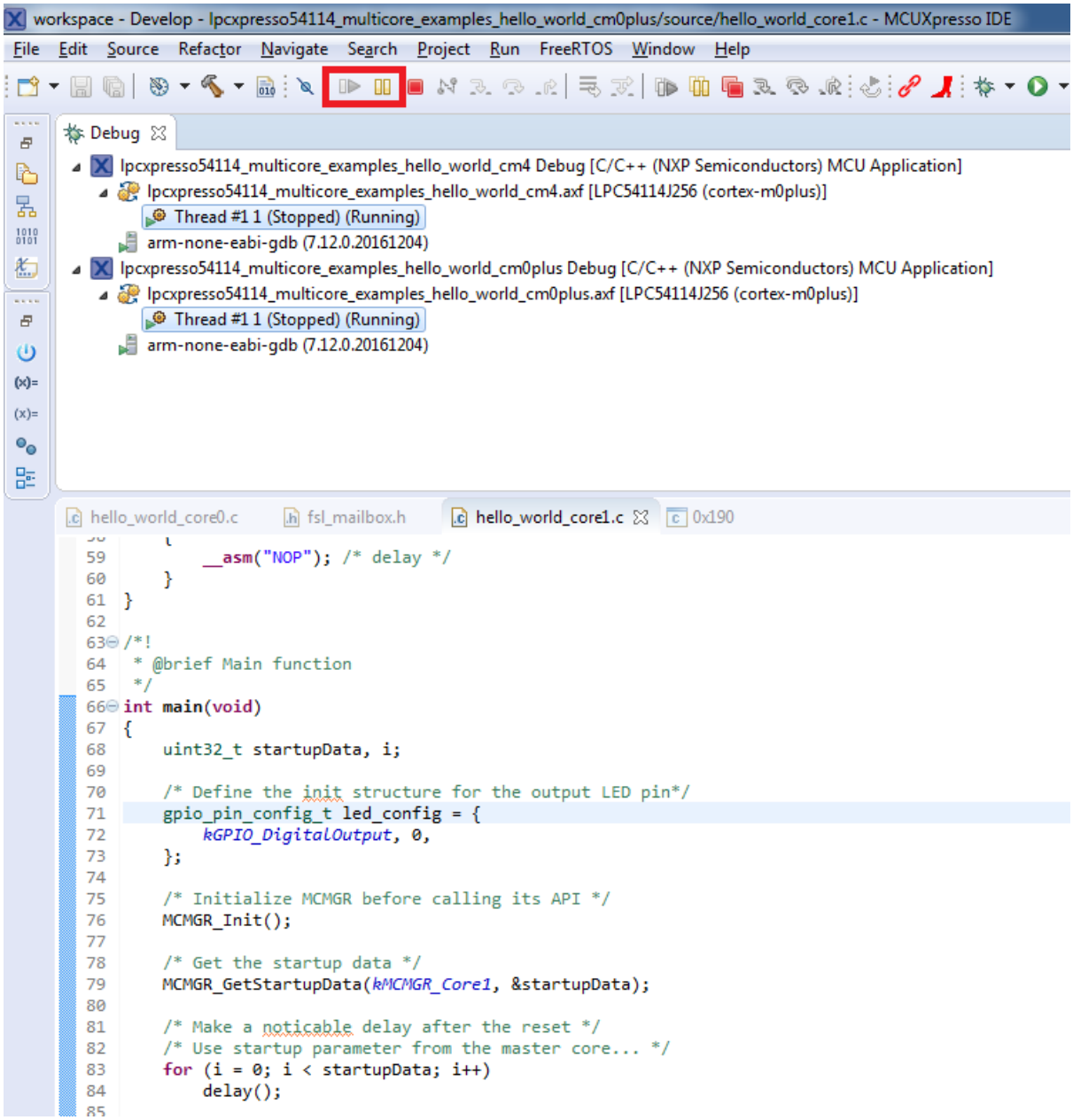

Quickstart Pa...
Global Varia...
Variables
Breakpoints
Outline
MCUXpresso IDE - Quickstart Panel
Project: lpcxpresso54114_hello_world_cm0plus [Debug]
Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean
Debug your project
Debug
Terminate, Build and Debug
Miscellaneous
Installed SDKs
Properties
Property
Debug using LinkServer probes (CTRL+SHIFT+L)
Attach to a running target using LinkServer (CTRL+ALT+L)
Program flash using LinkServer
Erase flash using LinkServer

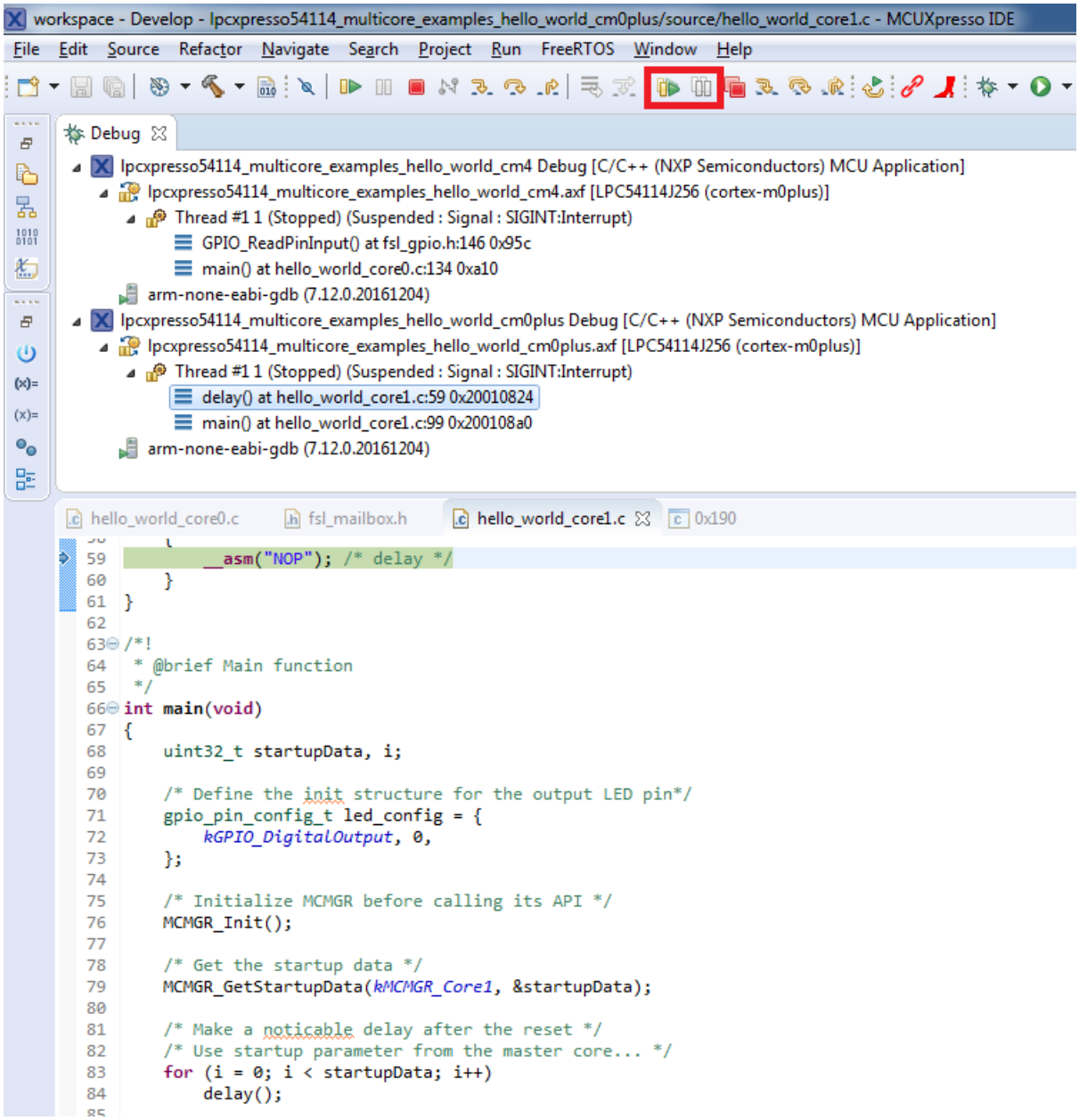


Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the “Resume” button. The hello_world multicore application then starts running. The primary core 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.



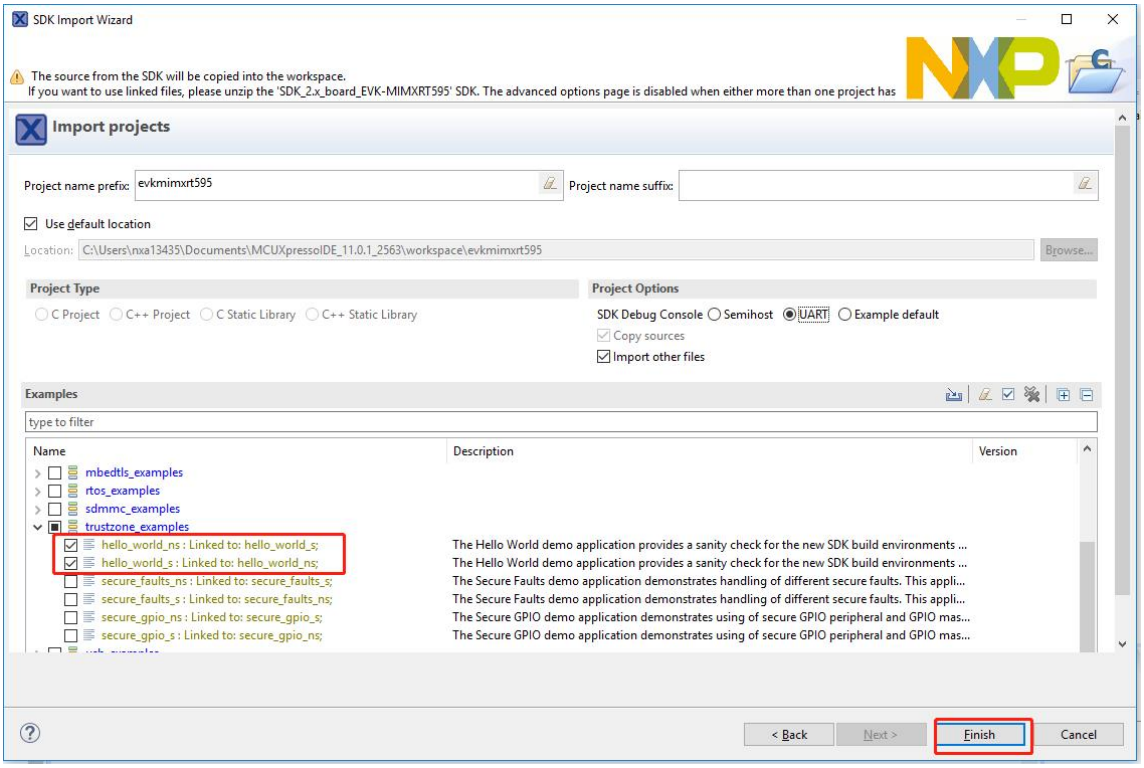
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the “Suspend” / “Resume” control button, or just using the “Suspend All Debug sessions” and the “Resume All Debug sessions” buttons.


workspace - Develop - lpcxpresso54114_multicore_examples_hello_world_cm0plus/source/hello_world_core1.c - MCUXpresso IDE
File Edit Source Refactor Navigate Search Project Run FreeRTOS Window Help
Debug
lpcxpresso54114_multicore_examples_hello_world_cm4 Debug [C/C++ (NXP Semiconductors) MCU Application]
lpcxpresso54114_multicore_examples_hello_world_cm4.axf [LPC54114J256 (cortex-m0plus)]
Thread #1 1 (Stopped) (Running)
arm-none-eabi-gdb (7.12.0.20161204)
lpcxpresso54114_multicore_examples_hello_world_cm0plus Debug [C/C++ (NXP Semiconductors) MCU Application]
lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf [LPC54114J256 (cortex-m0plus)]
Thread #1 1 (Stopped) (Running)
arm-none-eabi-gdb (7.12.0.20161204)
hello_world_core0.c
fsl_mailbox.h
hello_world_core1.c
0x190
59 __asm("NOP"); /* delay */
60 }
61 }
62
63 /*!
64 * @brief Main function
65 */
66 int main(void)
67 {
68 uint32_t startupData, i;
69
70 /* Define the init structure for the output LED pin*/
71 gpio_pin_config_t led_config = {
72 kGPIO_DigitalOutput, 0,
73 };
74
75 /* Initialize MCMGR before calling its API */
76 MCMGR_Init();
77
78 /* Get the startup data */
79 MCMGR_GetStartupData(kMCMGR_Core1, &startupData);
80
81 /* Make a noticable delay after the reset */
82 /* Use startup parameter from the master core... */
83 for (i = 0; i < startupData; i++)
84 delay();
85

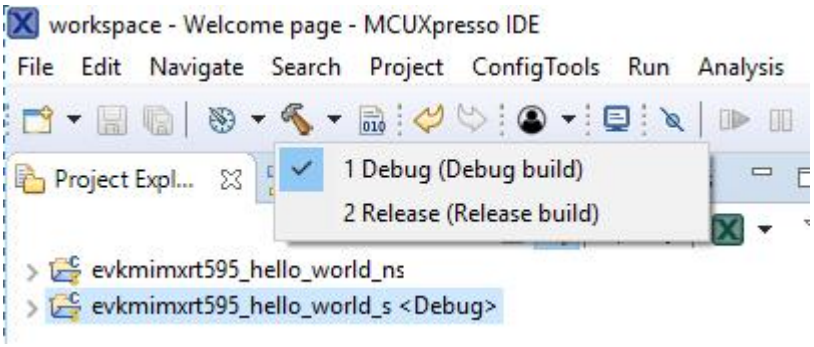


Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the hello_world example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though 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**.

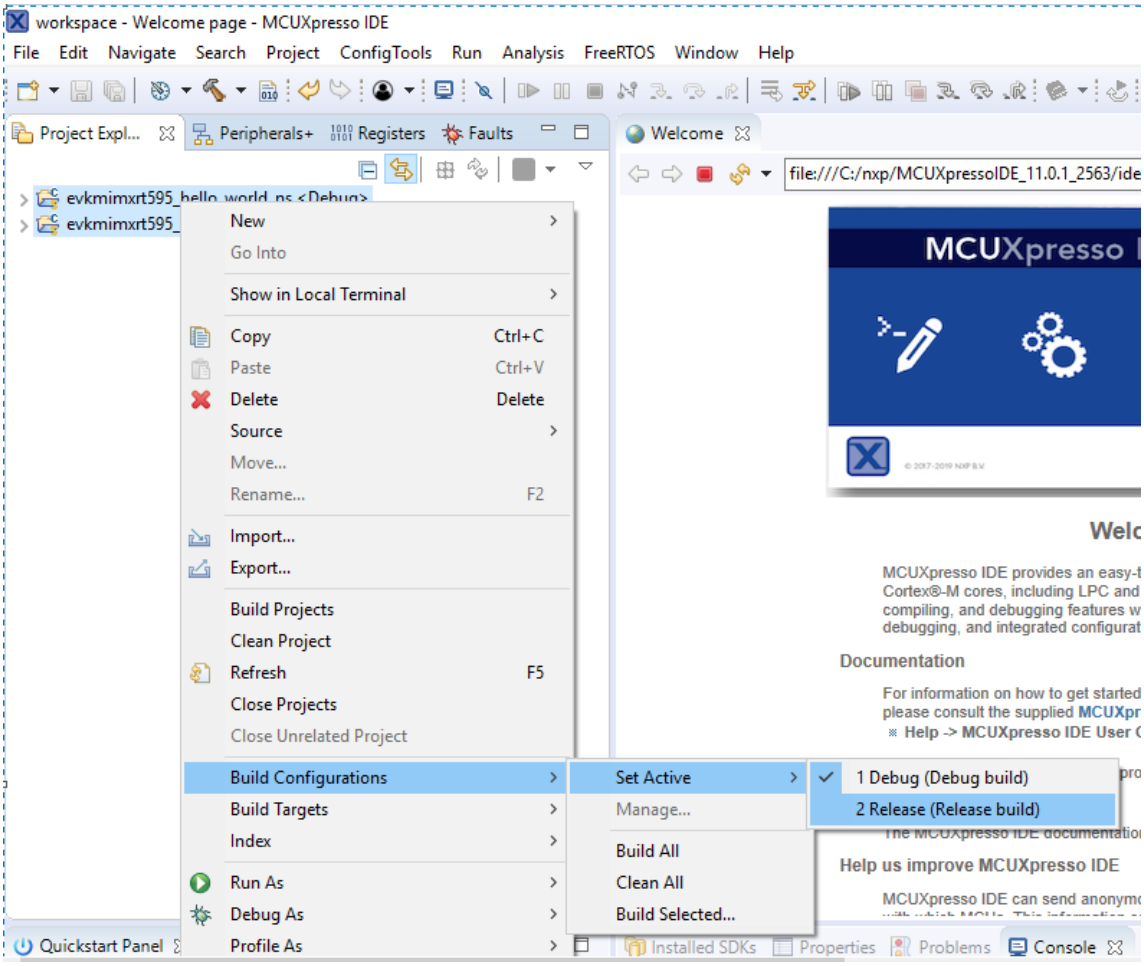


3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the evkmimxrt595_hello_world_s project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in following figure. For this example, select the **Debug** target.



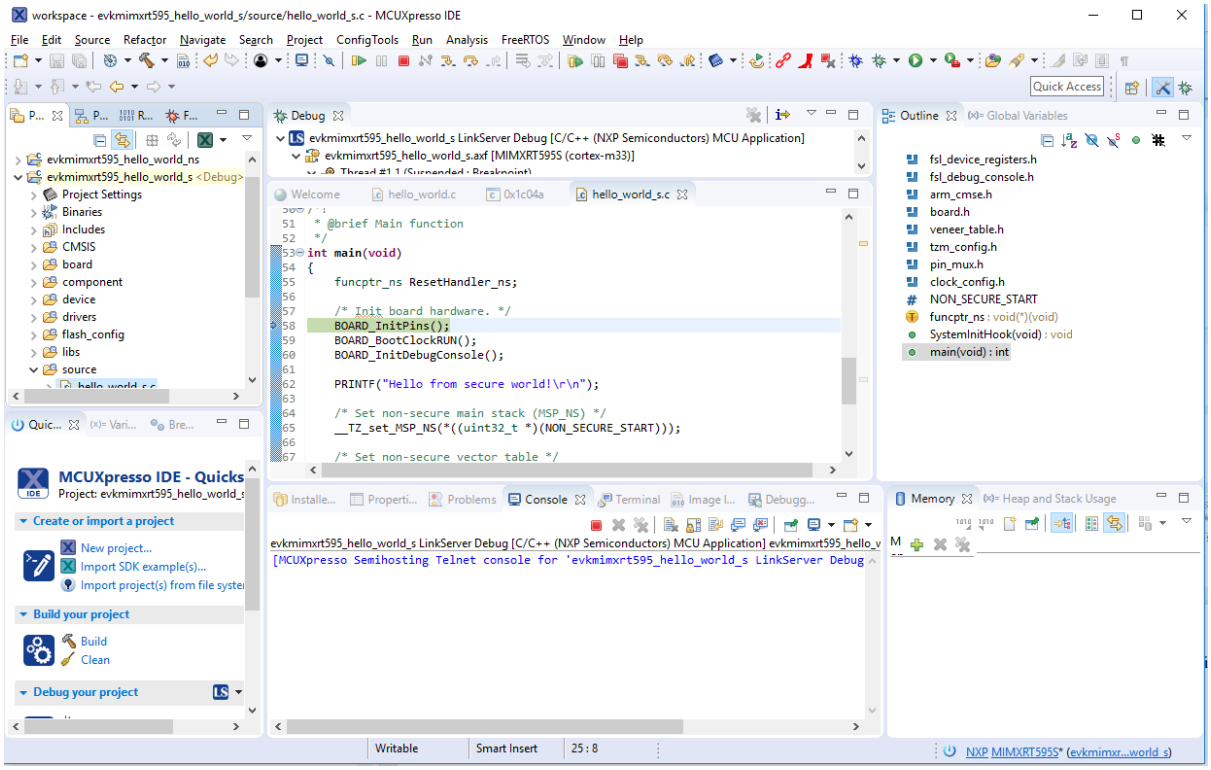
The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when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

Build an example application

Do the following steps to build the hello_world example application.

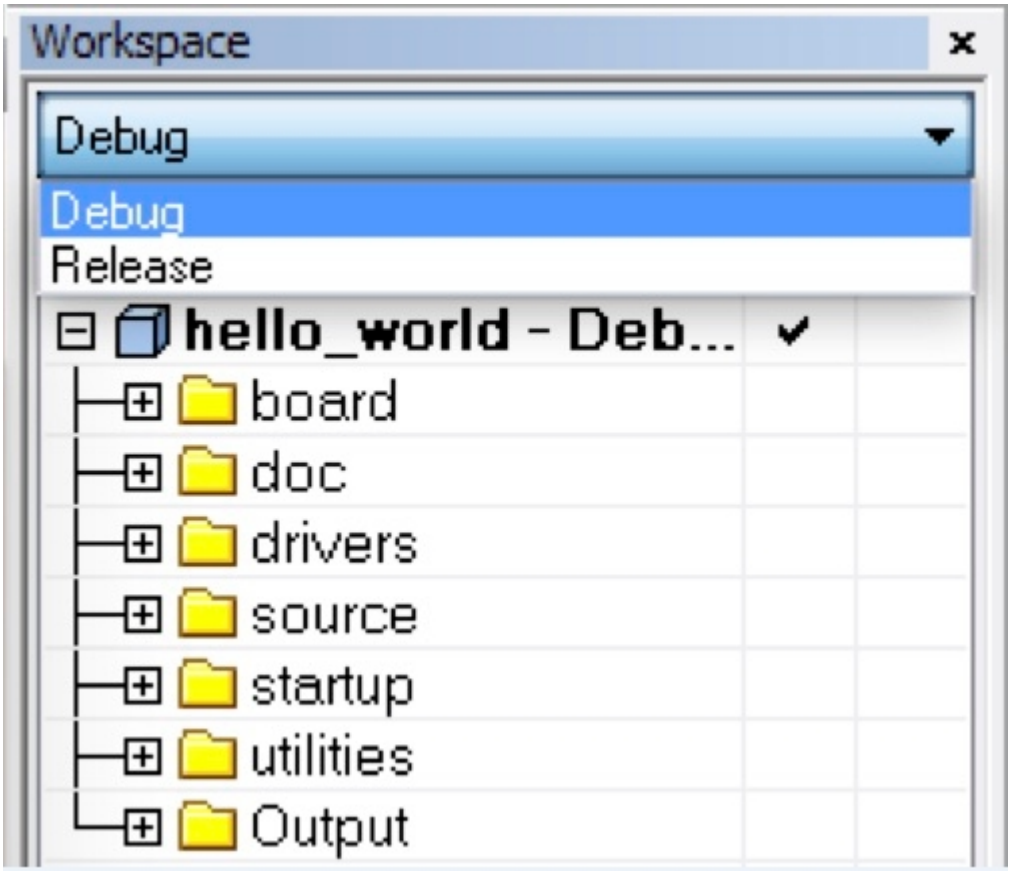
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

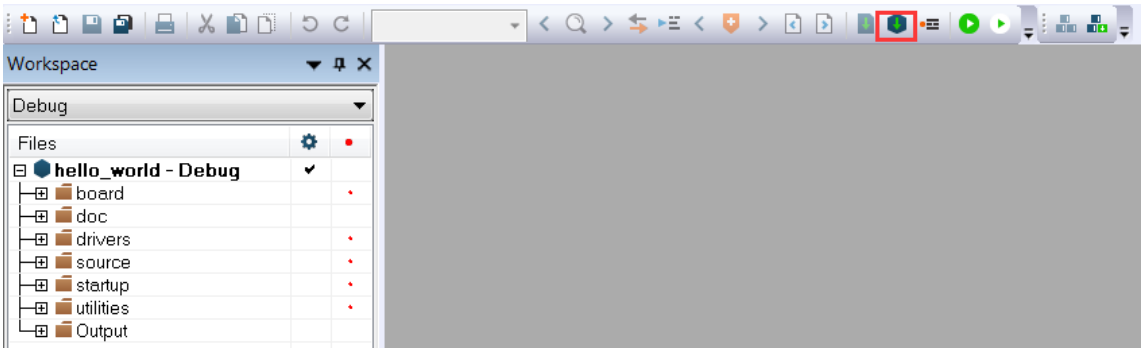
   Other example applications may have additional folders in their path.

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

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



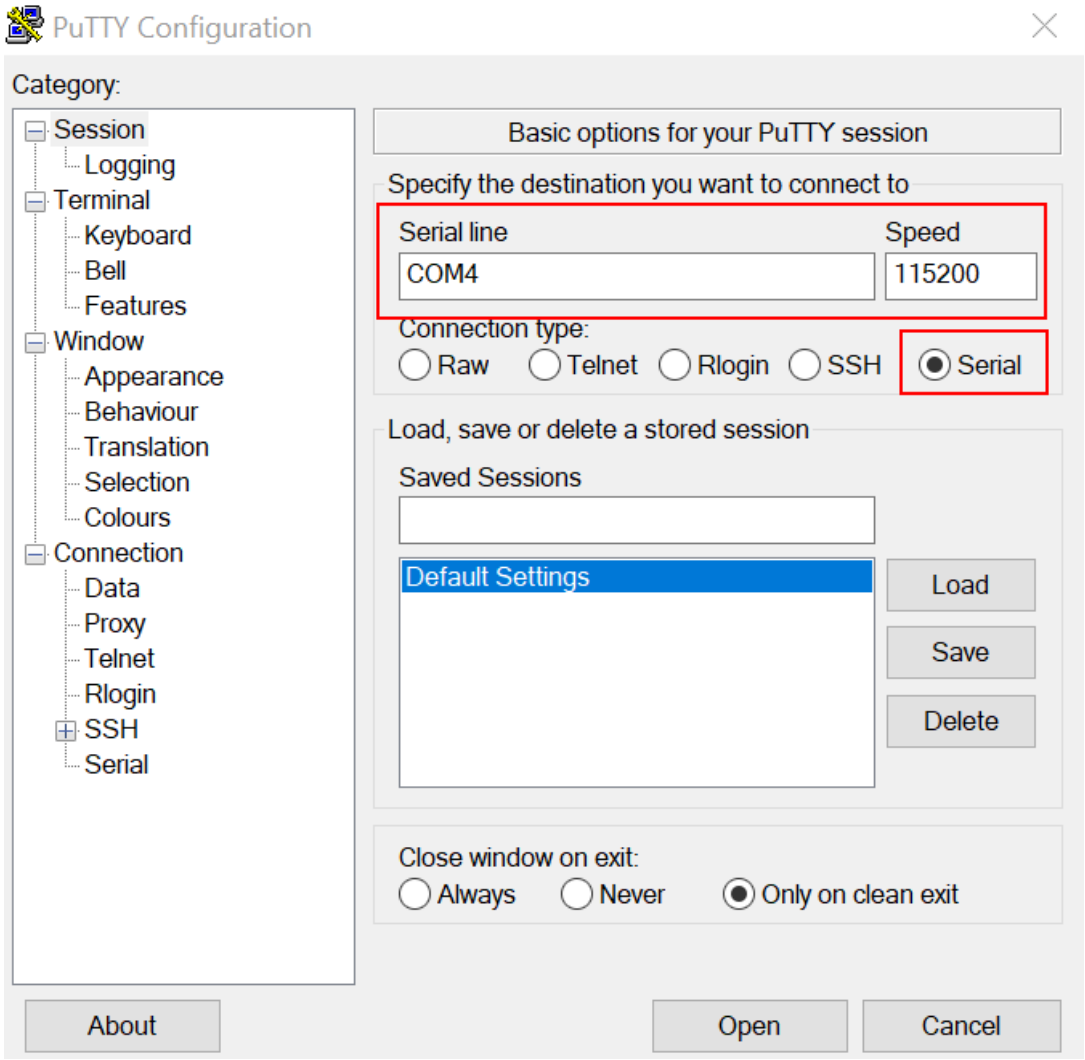
3. To build the demo application, click **Make**, highlighted in red in following figure.



4. The build completes without errors.

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

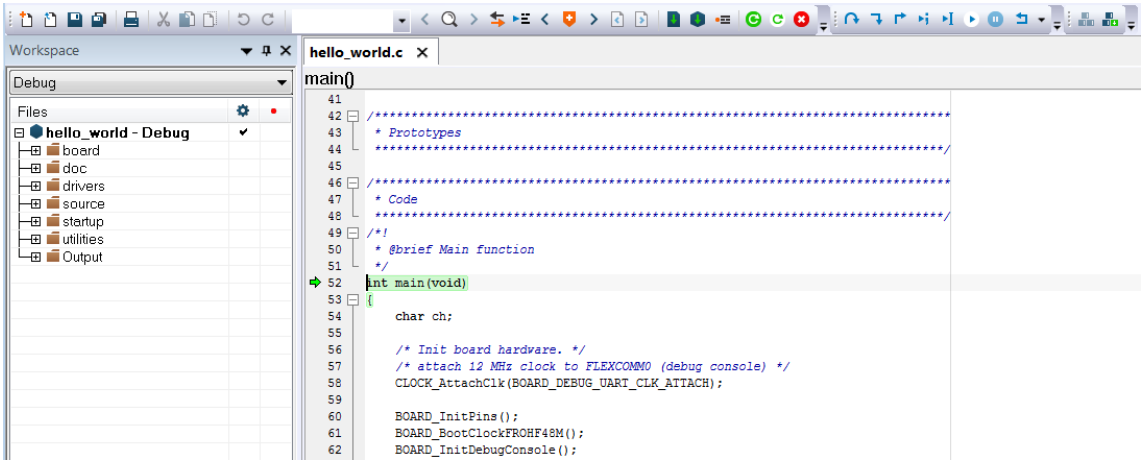
1. Ensure the host driver for the debugger firmware has been installed. See *On-board debugger*.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see *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

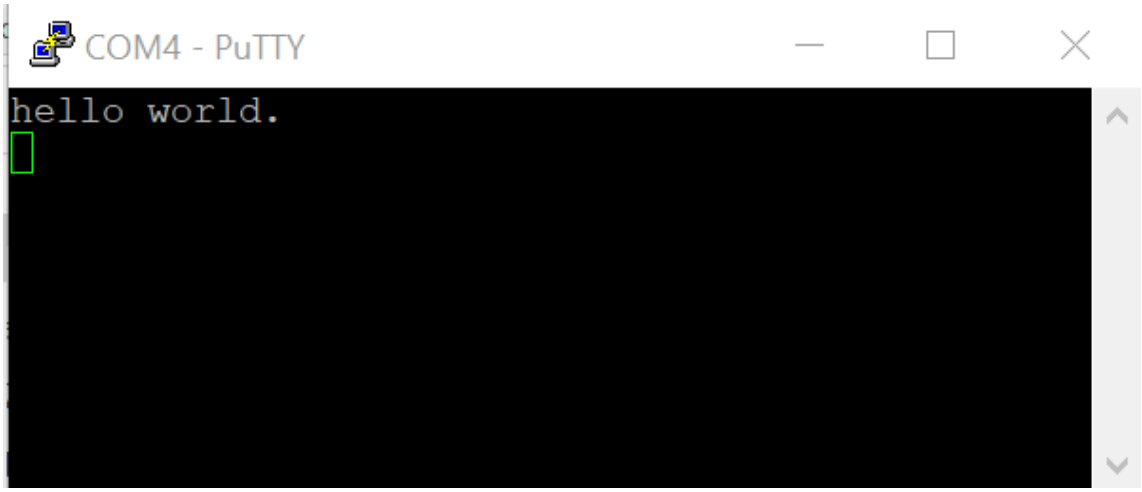
4. In IAR, click the **Download and Debug** button to download the application to the target.

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
```

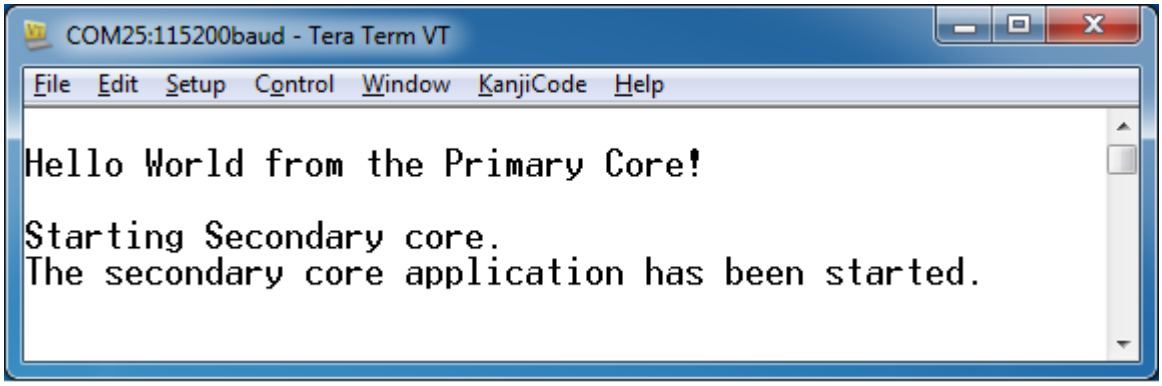
Build both applications separately by clicking the **Make** button. Build the application for the auxiliary core (cm0plus) first, because the primary core application project (cm4) must know the auxiliary core application binary when running the linker. It is not possible to finish the primary core linker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

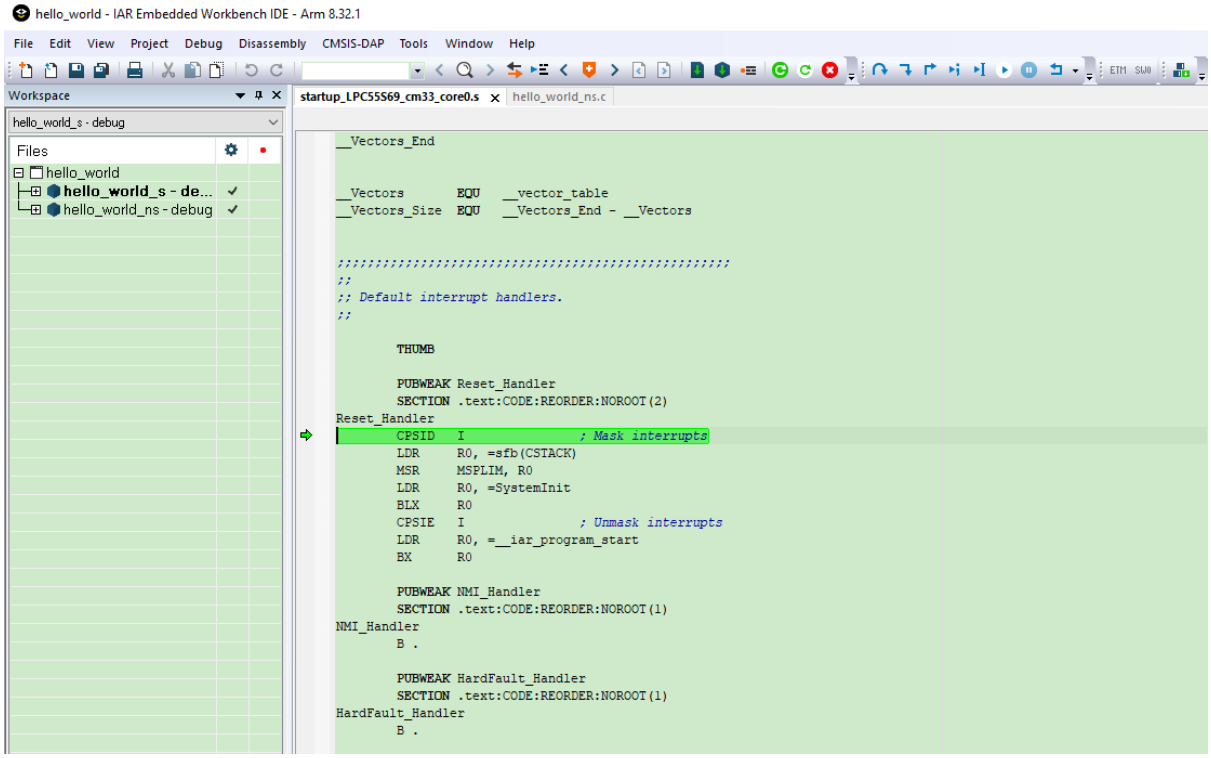
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_
↪ns.eww
```

```
<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
```

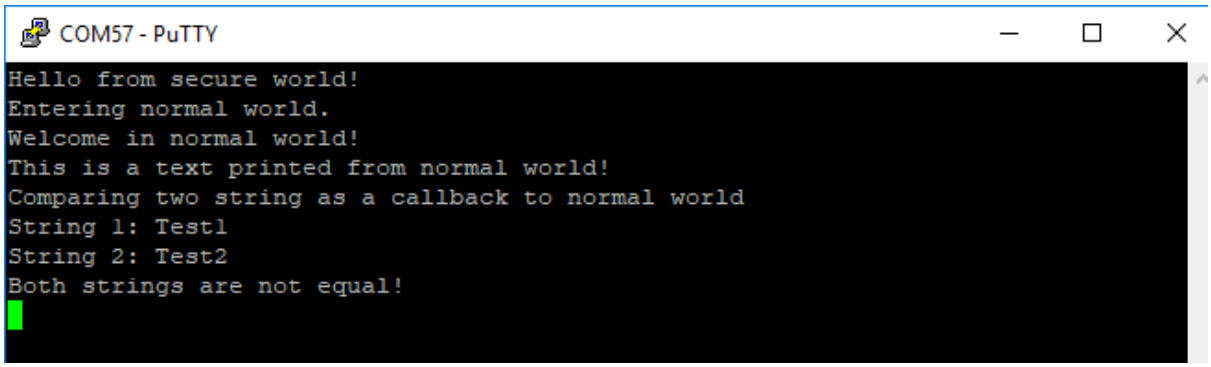
This project hello_world.eww contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because the 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.

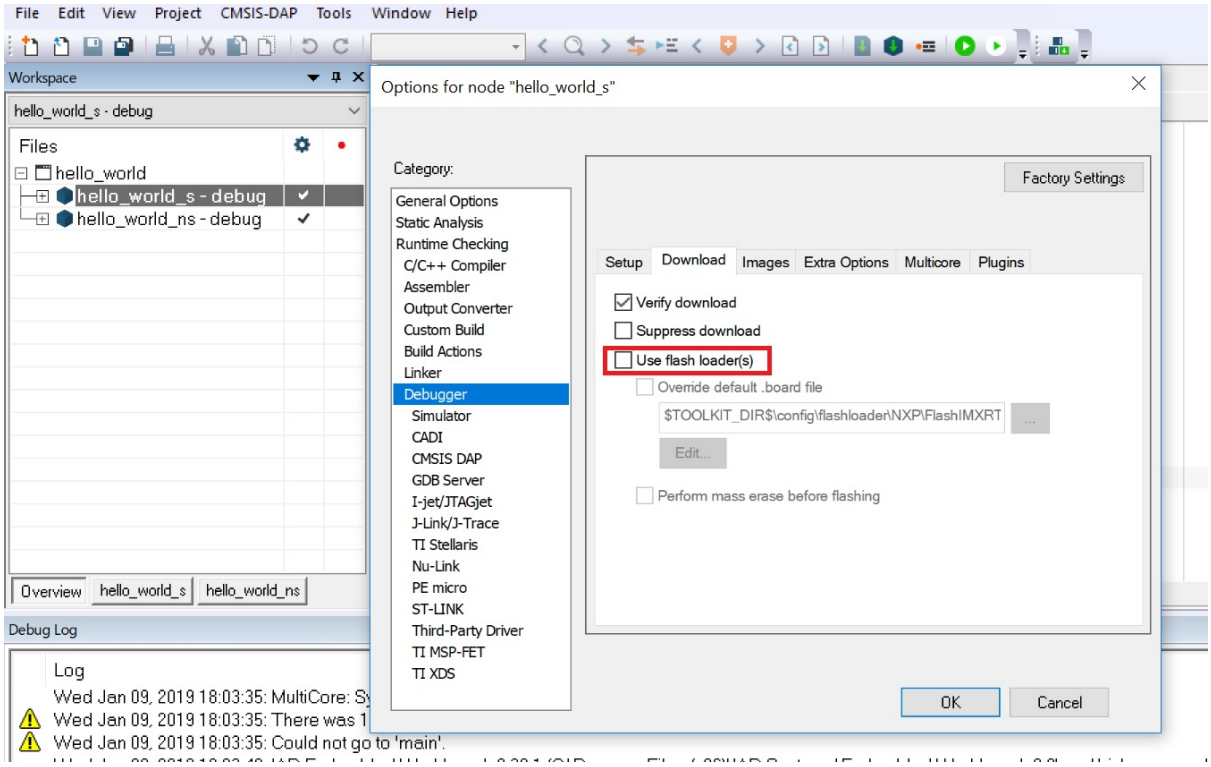


Run the code by clicking **Go** to start the application.

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.



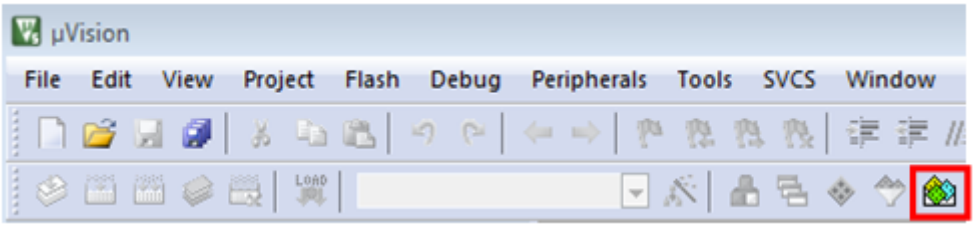
Note: If the application is running in RAM (debug/release build target), in **Options****>****Debugger** > **Download** tab, disable **Use flash loader(s)**. This can avoid the _ns download issue on i.MXRT500.



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
```

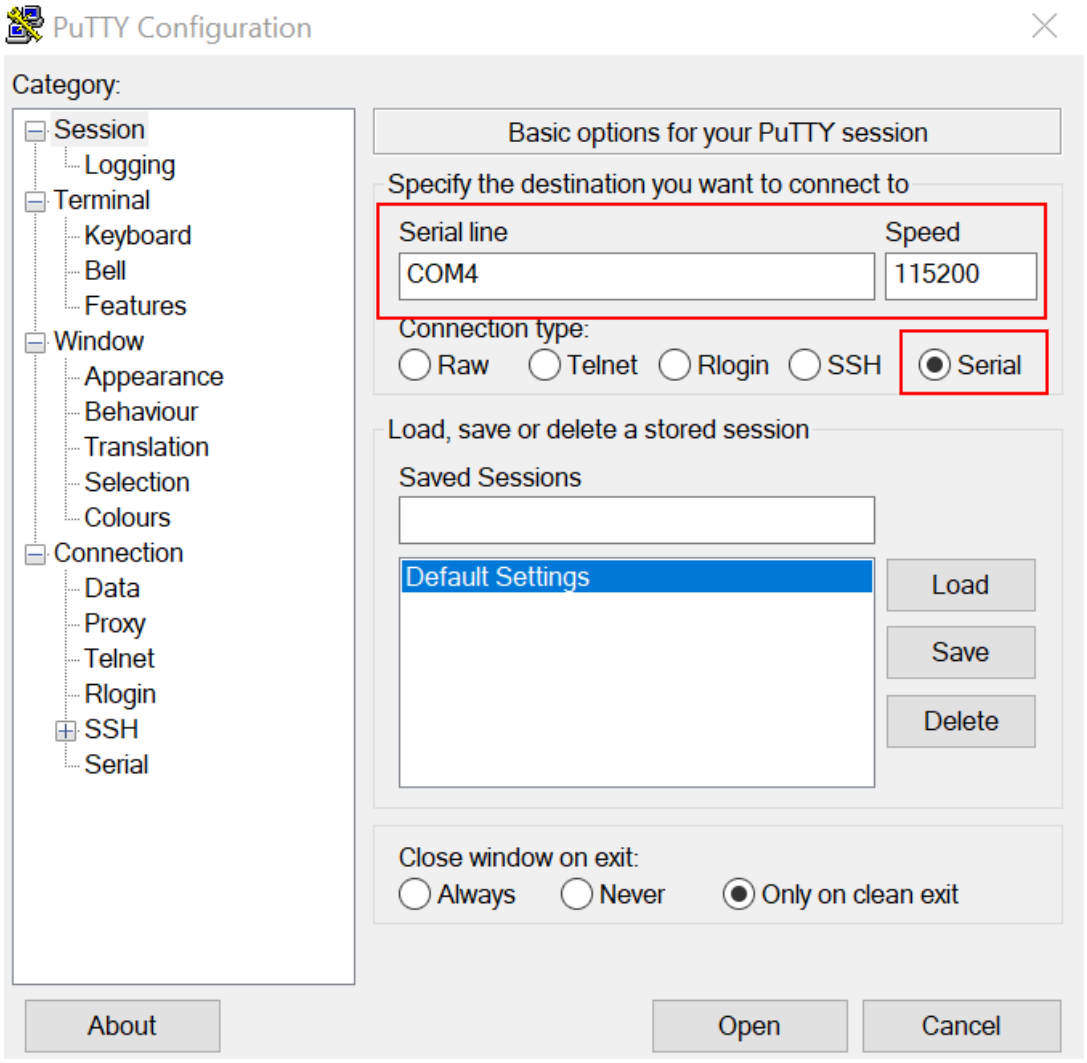
The workspace file is named as <demo_name>.uvmpw. For this specific example, the actual path is:

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

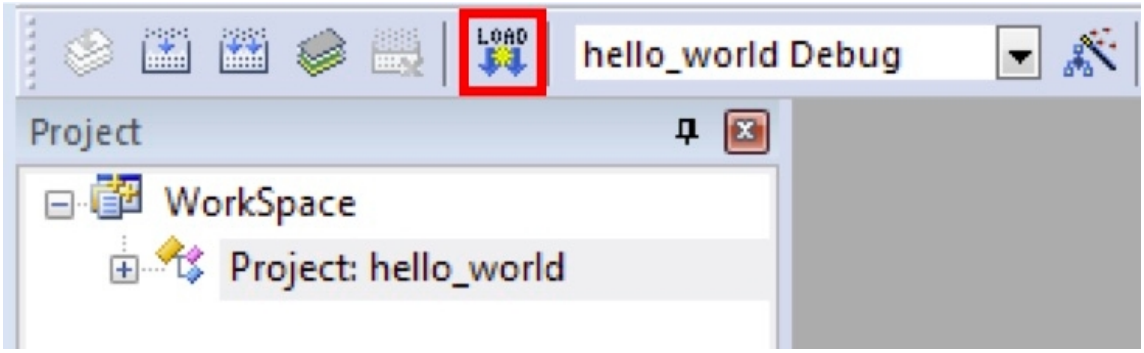
3. The build completes without errors.

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

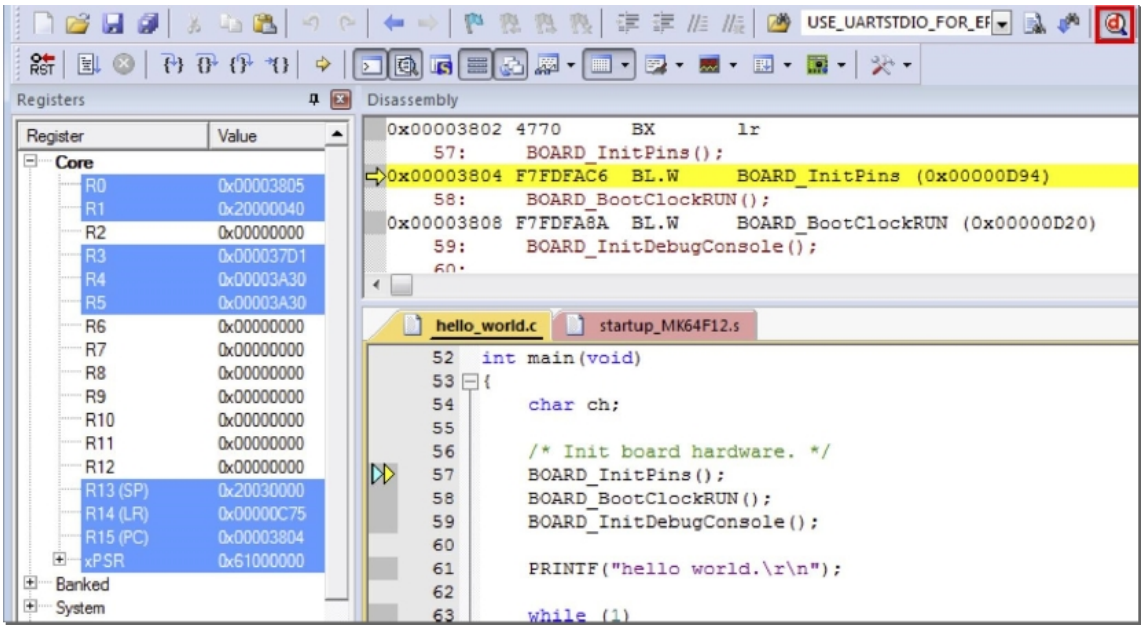
1. Ensure the host driver for the debugger firmware has been installed. See *On-board debugger*.
2. Connect the development platform to your PC via USB cable using USB connector.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm and connect to the debug serial port number (to determine 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



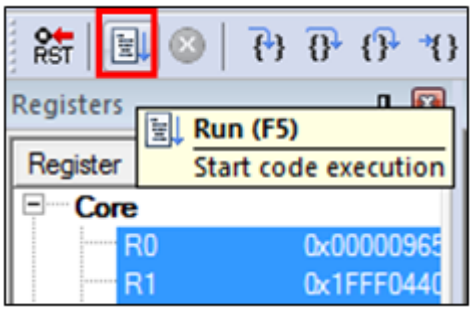
4. In µVision, after the application is built, click the **Download** button to download the application to the target.



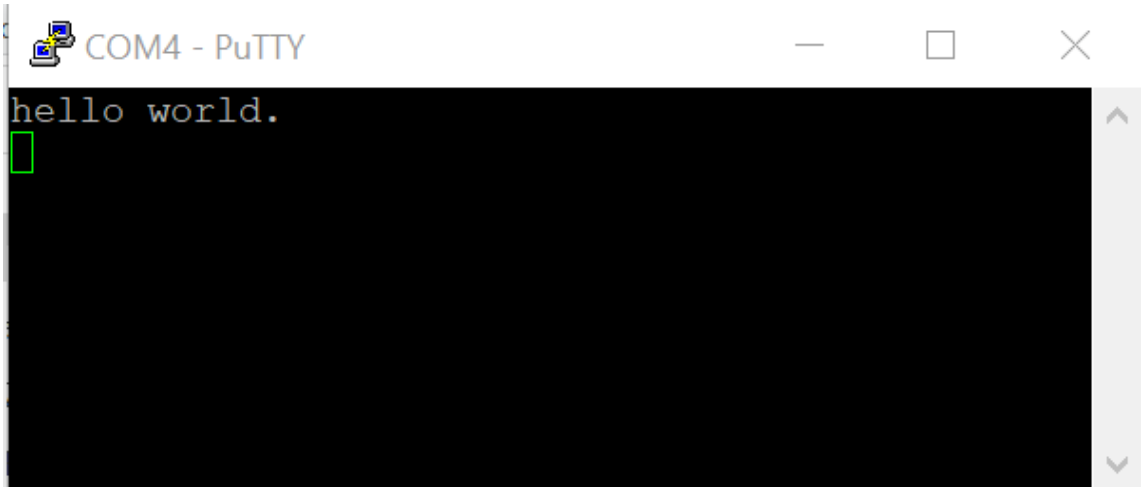
5. After clicking the **Download** button, the application downloads to the target and is running. To debug the application, click the **Start/Stop Debug Session** button, highlighted in red.



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:

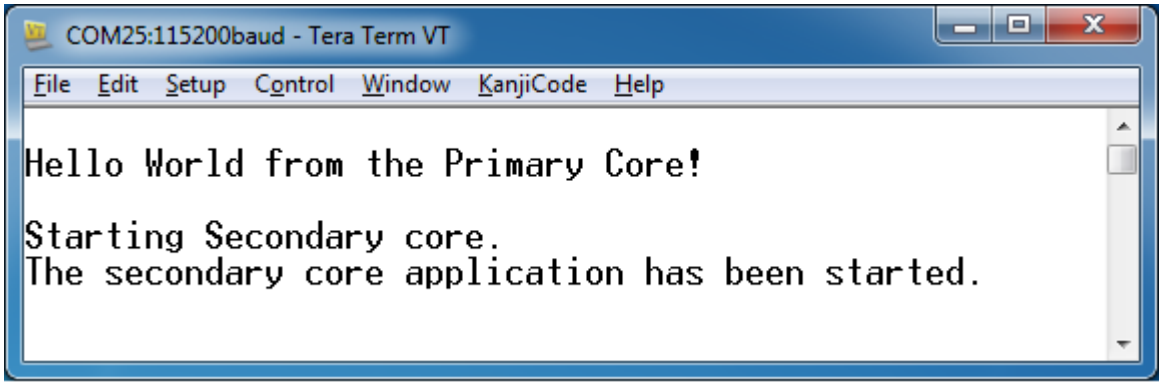
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
↪cm0plus.uvmpw
```

```
<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.

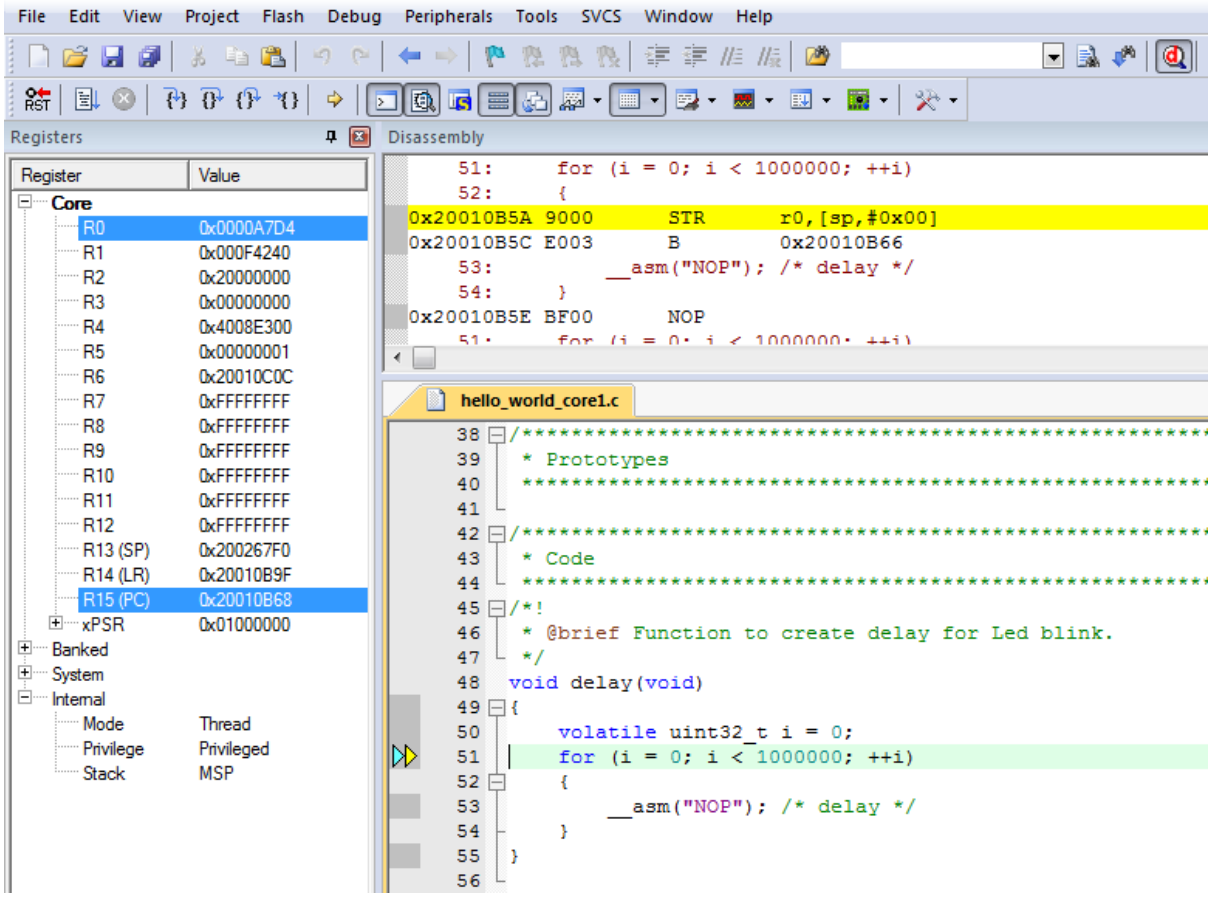
Run a multicore example application The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example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

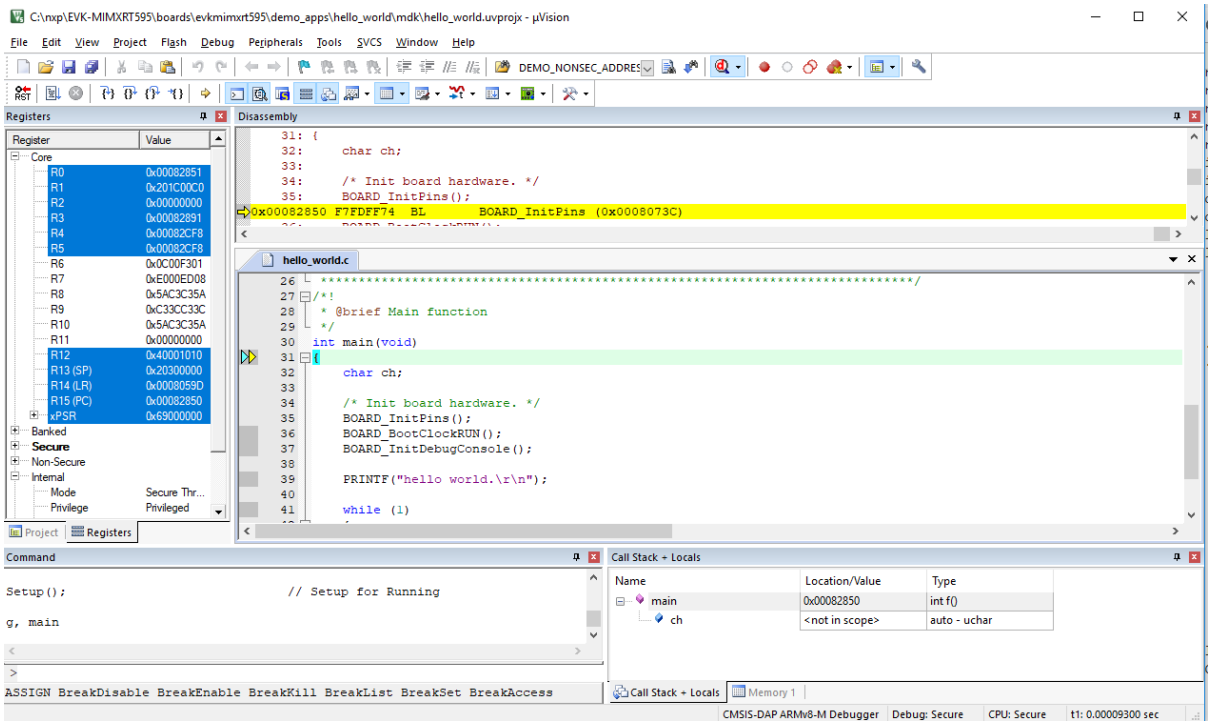
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.
↪uvmpw
```

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.

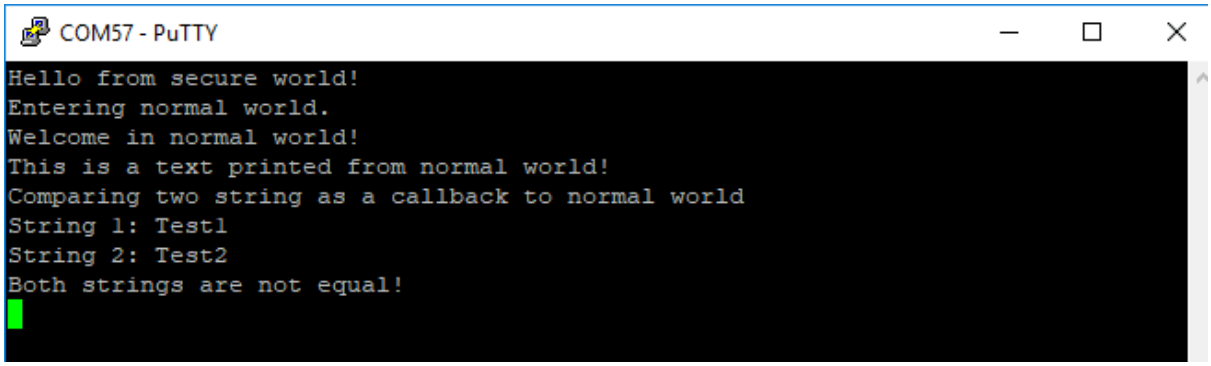
Run a TrustZone example application The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

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:

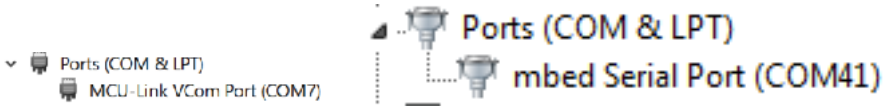
```
$ dmesg | grep "ttyUSB"
  [503175.307873] usb 3-12: cp210x converter now attached to ttyUSB0
  [503175.309372] usb 3-12: cp210x converter now attached to ttyUSB1
```

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

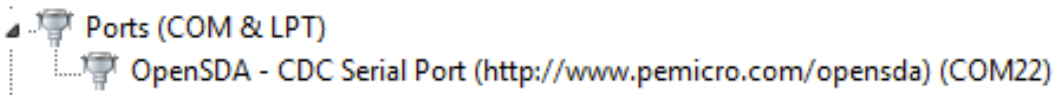
2. **Windows**: To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

   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.

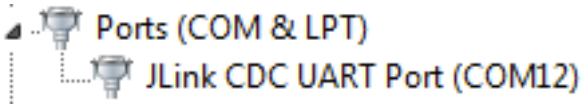
   1. **CMSIS-DAP/mbed/DAPLink** interface:

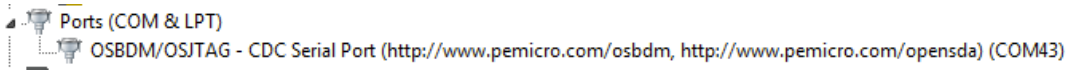
   2. **P&E Micro**:

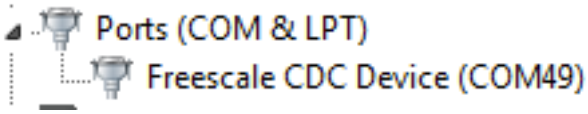
   3. **J-Link**:

   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.

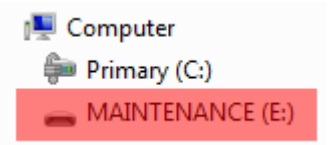
- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
- For boards with a P&E Micro interface, see PE micro to download 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

continues on next page

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

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

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

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

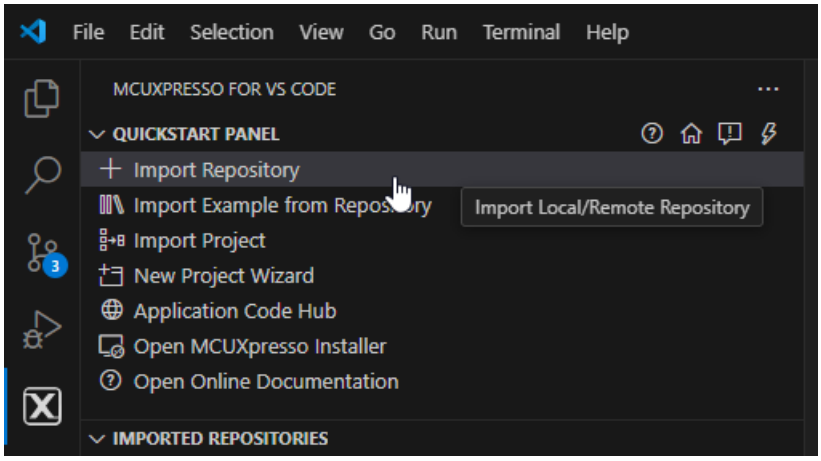
Extension Installation

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

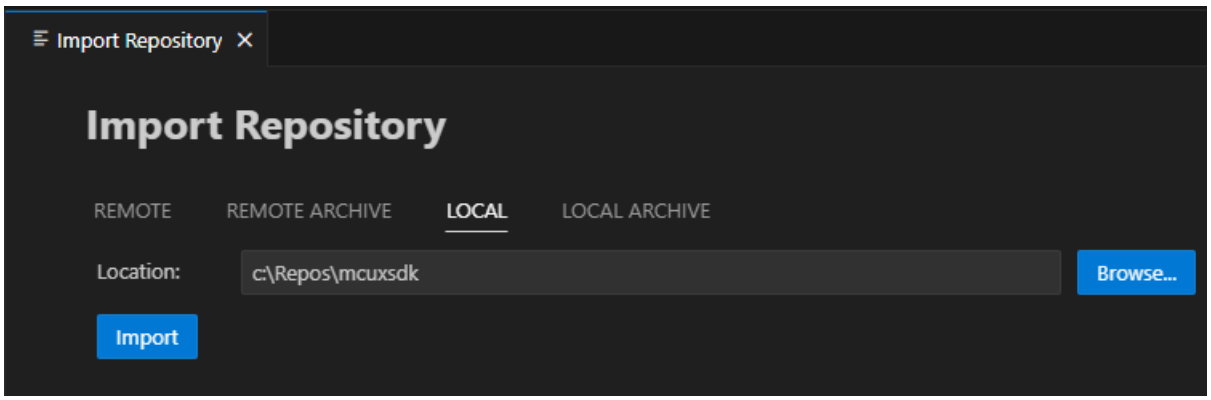
Import Methods The SDK can be imported in several ways. The MCUXpresso for VS Code extension supports both GitHub Repository SDK and Repository-Layout SDK Package distributions.

Import GitHub Repository SDK Click **Import Repository** from the **QUICKSTART PANEL**

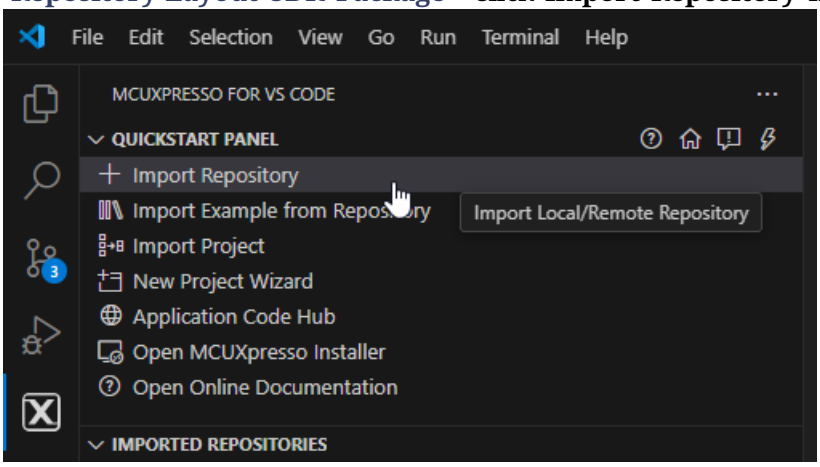


Note: You can import the SDK in several ways. Refer to MCUXpresso for VS Code Wiki for details.

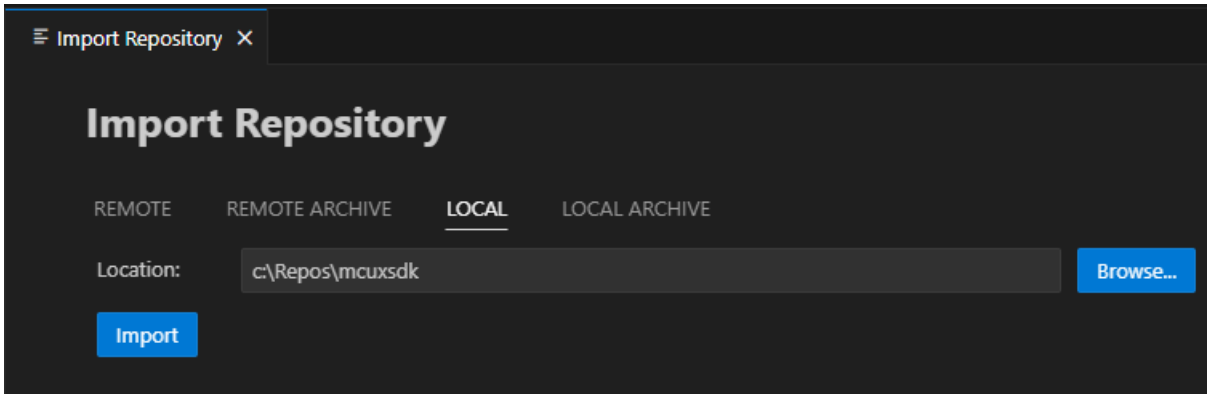
Select **Local** if you've already obtained the SDK according to *setting up the repo*. Select your location and click **Import**.



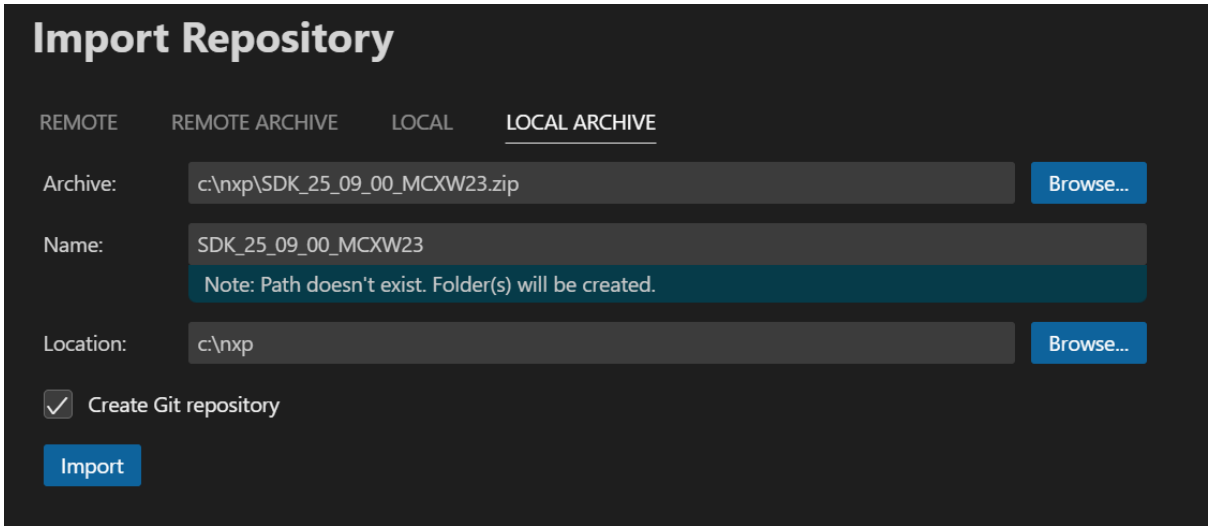
Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART PANEL**



Select **Local** if you've already unzipped the Repository-Layout SDK Package. Select your location and click **Import**.



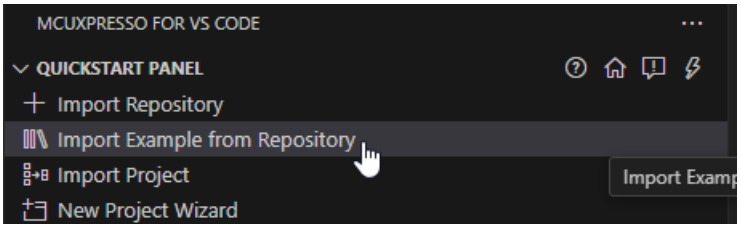
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



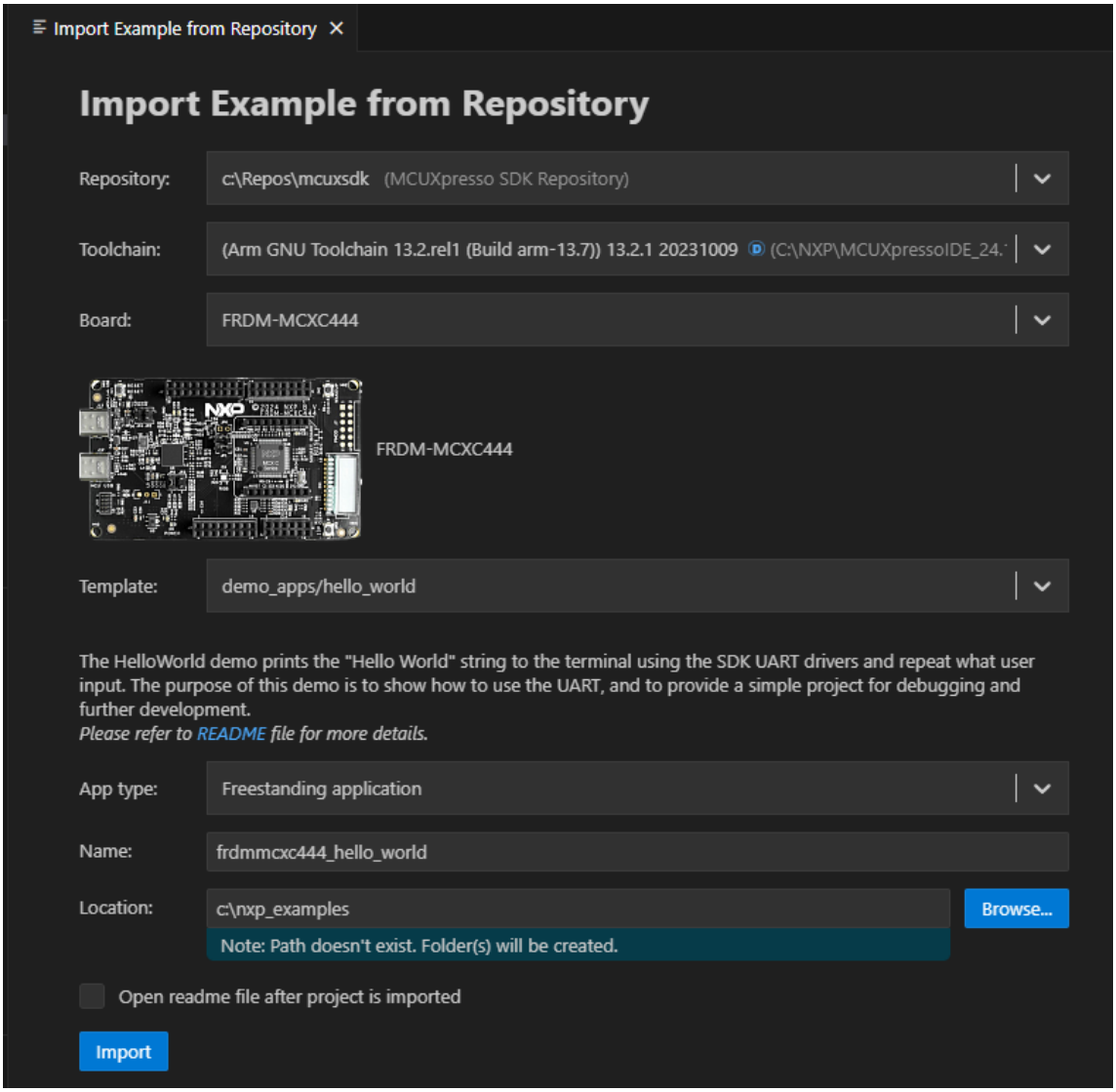
Building Example Applications

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:

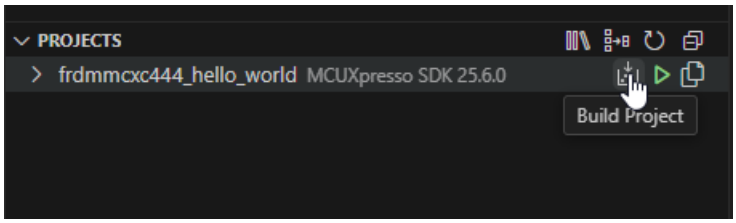
- Imported to user-defined location
- Independent of SDK location

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

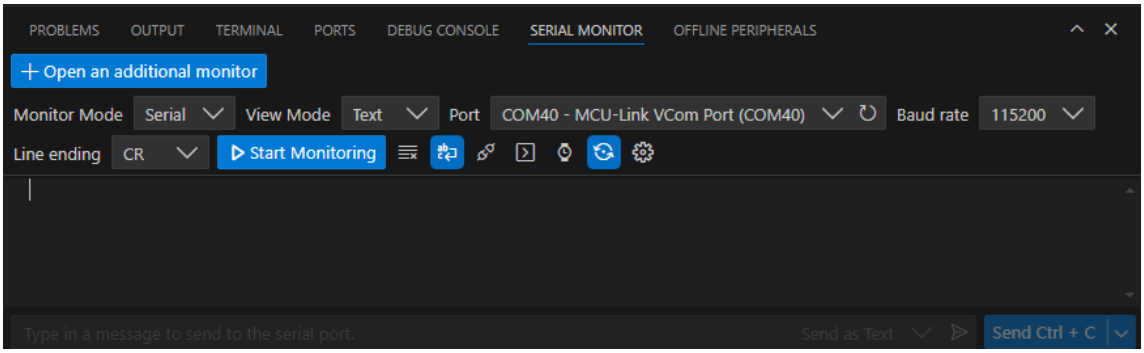


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

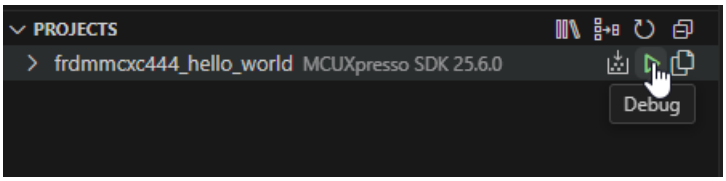
1. Open **Serial Monitor** from VS Code's integrated terminal



2. Configure serial settings:
   - **VCom Port**: Select port for your device
   - **Baud Rate**: Set to 115200

Debug Session

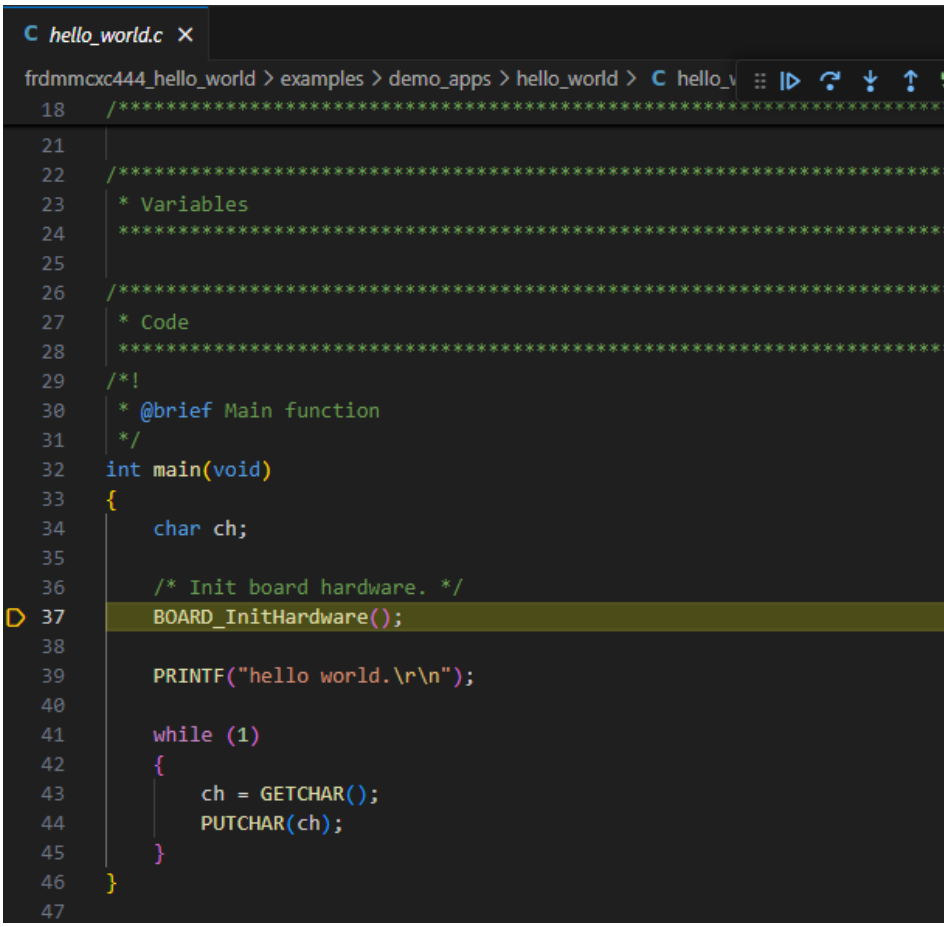
1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



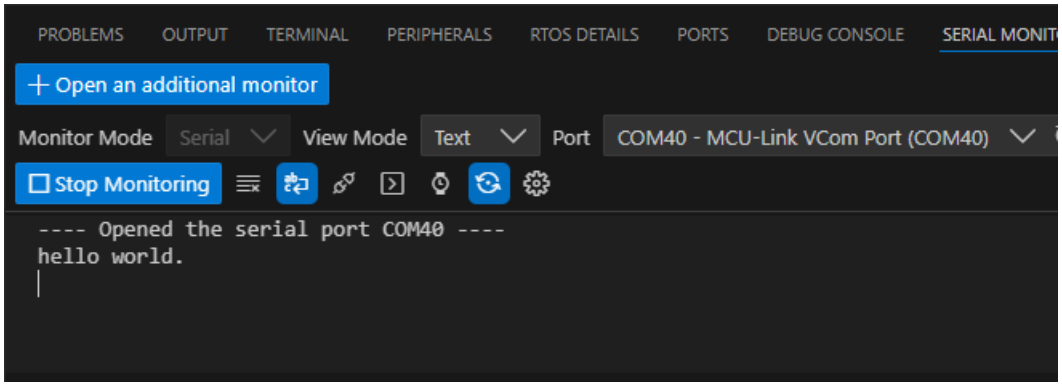
The debug session will begin with debug controls initially at the top of the interface.

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
LPCXpresso55	LPC5528JBD100, LPC5528JBD64, LPC5528JEV59, LPC5528JEV98, LPC5526JBD100, LPC5526JBD64, LPC5526JEV98, LPC55S26JBD100, LPC55S26JBD64, LPC55S26JEV98, **LPC55S28JBD100**, LPC55S28JBD64, LPC55S28JEV59, LPC55S28JEV98

MCUXpresso SDK release package

The MCUXpresso SDK release package content is aligned with the silicon subfamily it supports. This includes the boards, CMSIS, devices, middleware, and RTOS support.

Device support The device folder contains the whole software enablement available for the specific System-on-Chip (SoC) subfamily. This folder includes clock-specific implementation, device register header files, device register feature header files, and the system configuration source files. Included with the standard SoC support are folders containing peripheral drivers, toolchain support, and a standard debug console. The device-specific header files provide a direct access to the microcontroller peripheral registers. The device header file provides an overall SoC memory mapped register definition. The folder also includes the feature header file for each peripheral on the microcontroller. The toolchain folder contains the startup code and linker files for each supported toolchain. The startup code efficiently transfers the code execution to the main() function.

Board support The boards folder provides the board-specific demo applications, driver examples, and middleware examples.

Demo application and other examples The demo applications demonstrate the usage of the peripheral drivers to achieve a system level solution. Each demo application contains a readme file that describes the operation of the demo and required setup steps. The driver examples demonstrate the capabilities of the peripheral drivers. Each example implements a common use case to help demonstrate the driver functionality.

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

CMSIS DSP Library The MCUXpresso SDK is shipped with the standard CMSIS development pack, including the prebuilt libraries.

coreHTTP coreHTTP

PSA Test Suite Arm Platform Security Architecture Test Suite

mbedTLS mbedtls SSL/TLS library v3.x

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

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

TinyCBOR Concise Binary Object Representation (CBOR) Library

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

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

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.

NXP PSA CRYPTO DRIVER PSA crypto driver for crypto library integration via driver wrappers

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_nam
Utilities such as debug console	INSTALL_DIR/devices/<device_name>/utilities
Device Project Template for MCUXpresso IDE NPW	INSTALL_DIR/devices/<device_name>/project_templat
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.

safety_iec60730b cloned project fails to build

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

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

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version

ANACTRL

[2.4.0]

- Improvements
  - Added some interrupt flags for devices containing BOD1 and BOD2 interrupt controls.
  - Added a control macro to enable/disable the 32MHz Crystal oscillator code in current driver.
  - Added a feature macro for bit field ENA_96MHZCLK in FRO192M_CTRL.
  - Added a feature macro for bit field BODCORE_INT_ENABLE in BOD_DCDC_INT_CTRL.

[2.3.1]

- Bug Fixes
  - Added casts to prevent overflow caused by capturing large target clock.

[2.3.0]

- Improvements
  - Added AUX_BIAS control APIs.

[2.2.0]

- Improvements
  - Added some macros to separate the scenes that some bit fields are reserved for some devices.
  - Optimized the comments.
  - Optimized the code implementation inside some functions.

[2.1.2]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.3 and rule 17.7.

[2.1.1]

- Bug Fixes
  - Removed AnalogTestBus configuration to align with new header.

[2.1.0]

- Improvements
  - Updates for LPC55xx A1.
    * Removed the control of bitfield FRO192M_CTRL_ENA_48MHZCLK, XO32M_CTRL_ACBUF_PASS_ENABLE.
    * Removed status bits in ANACTRL_STATUS: PMU_ID OSC_ID FINAL_TEST_DONE_VECT.
    * Removed API ANACTRL_EnableAdcVBATDivider() and APIs which operate the RingOSC registers.
    * Removed the configurations of 32 MHz Crystal oscillator voltage source supply control register.
    * Added API ANACTRL_ClearInterrupts().

[2.0.0]

- Initial version.

CASSPER

[2.2.4]

- Fix MISRA-C 2012 issue.

[2.2.3]

- Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.

[2.2.2]

- Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE

[2.2.1]

- Fix MISRA C-2012 issue.

[2.2.0]

- Rework driver to support multiple curves at once.

[2.1.0]

- Add ECC NIST P-521 elliptic curve.

[2.0.10]

- Fix MISRA C-2012 issue.

[2.0.9]

- Remove unused function Jac_oncurve().
- Fix ECC384 build.

[2.0.8]

- Add feature macro for CASPER_RAM_OFFSET.

[2.0.7]

- Fix MISRA C-2012 issue.

[2.0.6]

- Bug Fixes
  - Fix IAR Pa082 warning

[2.0.5]

- Bug Fixes
  - Fix sign-compare warning

[2.0.4]

- For GCC compiler, enforce O1 optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires -fno-strict-aliasing.

[2.0.3]

- Bug Fixes
  - Fixed the bug for KPSDK-28107 RSUB, FILL and ZERO operations not implemented in enum _casper_operation.

[2.0.2]

- Bug Fixes
  - Fixed KPSDK-25015 CASPER_MEMCPY hard-fault on LPC55xx when both source and destination buffers are outside of CASPER_RAM.

[2.0.1]

- Bug Fixes
  - Fixed the bug that KPSDK-24531 double_scalar_multiplication() result may be all zeroes for some specific input.

[2.0.0]

- Initial version.

CLOCK

[2.3.8]

- Bug Fixes
  - Fixed an issue that ss_progmodfm_t, ss_progmoddp_t, and ss_modwvctrl_t use wrong shift value.

[2.3.7]

- Improvements
  - Add errata workaround for pll lock bit in CLOCK_SetPLL0Freq() and CLOCK_SetPLL1Freq().

[2.3.6]

- Bug Fixes
  - Correct the fail status condition in CLOCK_SetupExtClocking().

[2.3.5]

- Improvements
  - Added lost comments for some enumerations.

[2.3.4]

- Bug Fixes
  - Fix CLOCK_SetClkDiv function to set kCLOCK_DivFlexFrgx correctly.
  - Fixed violations of MISRA C-2012 rule 16.3, rule 10.4, rule 10.3.

[2.3.3]

- Bug Fixes
  - Fix the pllRate in CLOCK_SetPLL1Freq().

[2.3.2]

- Improvements
  - Optimized the CLOCK_GetPLL0OutFromSetup() function with the usage of input parameter.

[2.3.1]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.1, rule 12.2, rule 14.4 and so on.
  - Fixed IAR warning Pa082 for the clock driver.

[2.3.0]

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

[2.2.2]

- Bug Fixes
  - Corrected the PLL.SELI setting to align with new UM.
  - Changed the PLL lock reliable condition.

[2.2.1]

- Improvements
  - Removed redundant macro definitions.

[2.2.0]

- New Features
  - Added CLOCK_SetupPLUClkInClocking() to store the PLU CLKIN frequency.

[2.1.1]

- Improvements
  - Updated CLOCK_SetFLASHAccessCyclesForFreq() to support up to 150MHz frequency.

[2.1.0]

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

[2.0.4]

- Bug Fixes
  - Fixed C++ build errors in CLOCK_GetClockAttachId() and CLOCK_AttachClk().

[2.0.3]

- Bug Fixes
  - Fixed attach incorrect attach_id.

[2.0.2]

- New Features
  - Added get actual clock attach id API to allow users to obtain the actual clock source in target register.
- Bug Fixes
  - The attach clock and get actual clock attach id APIs should check combination of two clock source.
- Optimization
  - Made the judgement statements more clear.
  - Strengthened the compatibility of clock attach id.
  - Removed some non-meaningful definitions and add some useful ones to enhance readability.

[2.0.1]

- Some minor fixes.

[2.0.0]

- initial version.

CMP

[2.2.1]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.1, rule 10.4, and rule 17.7.

[2.2.0]

- Improvements:
  - Added API to configure the sampling mode and clock divider of the CMP Filter.
  - Supported CMP filter sampling mode configuration.

[2.1.0]

- New Features:
  - Added API to get default CMP user configuration structure.
  - Supported CMP filter clock divider settings.
  - Combined the settings of VREF source and VREF value into one API CMP_SetVREF().
  - Extracted CMP input source selection from CMP_Init() to CMP_SetInputChannels().
- Improvements:
  - Formatted API naming, variable naming and comment style for better readability.
  - Added comments for APIs in source file.

[2.0.1]

- Bug Fixes
  - Fixed missing 'const' qualifier for structure variable in function parameter.

[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_A 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.

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.

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.

HASHCRYPT

[2.0.0]

- Initial version.

[2.0.1]

- Supported loading AES key from unaligned address.

[2.0.2]

- Supported loading AES key from unaligned address for different compiler and core variants.

[2.0.3]

- Remove SHA512 and AES ICB algorithm definitions

[2.0.4]

- Add SHA context switch support

[2.1.0]

- Update the register name and macro to align with new header.
- Fixed the sign-compare warning in hashcrypt_load_data.

[2.1.1]

- Fix MISRA C-2012.

[2.1.2]

- Support loading AES input data from unaligned address.

[2.1.3]

- Fix MISRA C-2012.

[2.1.4]

- Fix context switch cannot work when switching from AES.

[2.1.5]

- Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to prevent possible optimization issue.

[2.2.0]

- Add AES-OFB and AES-CFB mixed IP/SW modes.

[2.2.1]

- Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() prevent compiler from reordering memory write when -O2 or higher is used.

[2.2.2]

- Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to fix optimization issue

[2.2.3]

- Added check for size in hashcrypt_aes_one_block to prevent overflowing COUNT field in MEMCTRL register, if its bigger than COUNT field do a multiple runs.

[2.2.4]

- In all HASHCRYPT_AES_xx functions have been added setting CTRL_MODE bitfield to 0 after processing data, which decreases power consumption.

[2.2.5]

- Add data synchronization barrier and instruction synchronization barrier inside hashcrypt_sha_process_message_data() to fix optimization issue

[2.2.6]

- Add data synchronization barrier inside HASHCRYPT_SHA_Update() and hashcrypt_get_data() function to fix optimization issue on MDK and ARMGCC release targets

[2.2.7]

- Add data synchronization barrier inside HASHCRYPT_SHA_Update() to fix optimization issue on MCUX IDE release target

[2.2.8]

- Unify hashcrypt hashing behavior between aligned and unaligned input data

[2.2.9]

- Add handling of set ERROR bit in the STATUS register

[2.2.10]

- Fix missing error statement in hashcrypt_save_running_hash()

[2.2.11]

- Fix incorrect SHA-256 calculation for long messages with reload

[2.2.12]

- Fix hardfault issue on the Keil compiler due to unaligned memcpy() input on some optimization levels

[2.2.13]

- Added function hashcrypt_seed_prng() which loading random number into PRNG_SEED register before AES operation for SCA protection

[2.2.14]

- Modify function hashcrypt_get_data() to prevent issue with unaligned access

[2.2.15]

- Add wait on DIGEST BIT inside hashcrypt_sha_one_block() to fix issues with some optimization flags

[2.2.16]

- Add DSB instruction inside hashcrypt_sha_ldm_stm_16_words() to fix issues with some optimization flags

[2.2.17]

- Fix context size when hashcrypt built with reload feature

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 kReceiveDataBeginState 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_TransferReceiveLoopD 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

[2.1.5]

- Improvements
  - Update Flash_Program src parameter to const.
  - Check CPU frequency <= 100MHZ for Flash Erase and Program.
  - Add BOOTLOADER_UserEntry API.

[2.1.4]

- Bug Fixes
  - Fixed misra issue.

[2.1.3]

- Bug Fixes
  - Fix the CFPA version wasn't transferred into SDK driver.

[2.1.2]

- Bug Fixes
  - Fix IAP driver status definitions don't match ROM_API.pdf from User Manual.

[2.1.1]

- Bug Fixes
  - The last 17 pages are reserved for chips with 640KB flash.

[2.1.0]

- New Features
  - Added new API FLASH_Read for users to read flash.
  - Added new API skboot_authenticate for image authentication api.
  - Added new AP kb_init, kb_deinit, kb_execute for users to operate BOOT ROM.

[2.0.3]

- Bug Fixes
  - Resolve incompatibility issue.

[2.0.2]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 11.1.
- Improvements
  - Improved the format of IAP driver version, using versionMajor to obtain the major version of bootloader.

[2.0.1]

- Improvements
  - Removed the enumeration item kSysToFlashFreq_100MHz which cannot be supported.
  - Removed the invalid FFR commands.
  - Improved the format of IAP driver version, using S_VersionMajor to obtain the major version of bootloader.

[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 'IOCON_GPIO_MODE'. The original is kept as a deprecated symbol.

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.

LPADC

[2.9.5]

- Improvements
  - Fix doxygen issue, grouping command should be balanced.

[2.9.4]

- Improvements
  - Update LPADC_GetDefaultConfig, change default conversionAverageMode value to: kLPADC_ConversionAverage128 for 3 bit width. kLPADC_ConversionAverage1024 for 4 bit width.

[2.9.3]

- Improvements
  - Add timeout for while loop code.

[2.9.2]

- Improvements
  - Fixed CERT-C issues.

[2.9.1]

- Bug Fixes
  - Fixed incorrect channel B FIFO selection logic.

[2.9.0]

- Bug Fixes
  - Add code to handle the case where GCC[GAIN_CAL] is a signed number.
  - Split LPADC_FinishAutoCalibration function into two functions.
  - Improved LPADC driver.

[2.8.4]

- Bug Fixes
  - Remove function 'LPADC_SetOffsetValue' assert statement, this statement may cause runtime errors in existing code.

[2.8.3]

- Bug Fixes
  - Fixed SDK lpadc driver examples compile issue, move condition 'commandId < ADC_CV_COUNT' to a more appropriate location.

[2.8.2]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 18.1, 10.3, 10.1 and 10.4.

[2.8.1]

- Bug Fixes
  - Fixed LPADC sample mode enum name mistake.

[2.8.0]

- Improvements
  - Release peripheral from reset if necessary in init function.
- Bug Fixes
  - Fixed function LPADC_GetConvResult() issue.
  - Fixed function LPADC_SetConvCommandConfig() bugs.

[2.7.2]

- Improvements
  - Use feature macros instead of header file macros.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 10.1, 10.3, 10.4 and 14.3.

[2.7.1]

- Improvements
  - Corrected descriptions of several functions.
  - Improved function LPADC_GetOffsetValue and LPADC_SetOffsetValue.
  - Revert changes of feature macros for lpadc.
  - Use feature macros instead of header file macros.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 10.8.
  - Fixed the violations of MISRA C-2012 rule 10.1, 10.3, 10.4 and 14.3.

[2.7.0]

- Improvements
  - Added supports of CFG2 register.
  - Removed some useless macros.

[2.6.2]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules.
  - Fixed LPADC driver code compile error issue.

[2.6.1]

- Improvements
  - Updated the use of macros in the driver code.

[2.6.0]

- Improvements
  - Added the API LPADC_SetOffset12BitValue() to configure 12bit ADC conversion offset trim value manually.
  - Added the API LPADC_SetOffset16BitValue() to configure 16bit ADC conversion offset trim value manually.
  - Added API to set offset calibration mode.
  - Added configuration of alternate channel.
  - Updated auto calibration API and added calibration value conversion API.
- New feature
  - Added API LPADC_EnableHardwareTriggerCommandSelection() to enable trigger commands controlled by ADC_ETC.
  - Updated LPADC_DoAutoCalibration() to allow doing something else before the ADC initialization to be totally complete. Enhance initialization duration time of the ADC.
  - Added two new APIs to get/set calibration value.

[2.5.2]

- Improvements
  - Added while loop, LPADC_GetConvResult() will return only when the FIFO will not be empty.

[2.5.1]

- Bug Fixes
  - Fixed some typos in Lpadc driver comments.

[2.5.0]

- Improvements
  - Added missing items to enable trigger interrupts.

[2.4.0]

- New features
  - Added APIs to get/clear trigger status flags.

[2.3.0]

- Improvements
  - Removed LPADC_MeasureTemperature() function for the LPADC supports different temperature sensor calculation equations.

[2.2.1]

- Improvements
  - Optimized LPADC_MeasureTemperature() function to support the specific series with flash solidified calibration value.
  - Clean doxygen warnings.
- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.3, rule 10.8 and rule 17.7.

[2.2.0]

- New Feature
  - Added API LPADC_MeasureTemperature() to get correct temperature from the internal sensor.
- Improvements
  - Separated lpadc_conversion_resolution_mode_t with related feature macro.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules:
    - Rule 10.3, 10.4, 10.6, 10.7 and 17.7.

[2.1.1]

- Improvements
  - Updated the gain calibration formula.
  - Used feature to segregate the new item kLPADC_TriggerPriorityPreemptSubsequently.

[2.1.0]

- New Features
  - Added the API LPADC_SetOffsetValue() to support configure offset trim value manually.
  - Added the API LPADC_DoOffsetCalibration() to do offset calibration independently.
- Improvements
  - Improved the usage of macros and removed invalid macros.

[2.0.2]

- Improvements
  - Added support for platforms with 2 FIFOs and different calibration measures.

[2.0.1]

- Bug Fixes
  - Ensured the API LPADC_SetConvCommandConfig configure related registers correctly.

[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.

OSTIMER

[2.2.6]

- Improvements
  - Drop the check of MATCH_WR_RDY and ostimer counter value in OSTIMER_SetMatchRawValue. In most applications, they are useless and may bring at least 7 OSTimer ticks latency, which is unacceptable when OSTimer is working under slow peripheral clock.
  - Optimize software gray code to binary conversion: replaced loop-based implementation with branchless bitwise operations.

[2.2.5]

- Improvements
  - Support binary encoded ostimer.

[2.2.4]

- Bug Fixes
  - Fixed CERT INT31-C violations.

[2.2.3]

- Improvements
  - Disable and clear pending interrupts before disabling the OSTIMER clock to avoid interrupts being executed when the clock is already disabled.

[2.2.2]

- Improvements
  - Support devices with different OSTIMER instance name.

[2.2.1]

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

[2.2.0]

- Improvements
  - Move the PMC operation out of the OSTIMER driver to board specific files.
  - Added low level APIs to control OSTIMER MATCH and interrupt.

[2.1.2]

- Bug Fixes
  - Fixed MISRA-2012 rule 10.8.

[2.1.1]

- Bug Fixes
  - removes the suffix 'n' for some register names and bit fields' names
- Improvements
  - Added HW CODE GRAY feature supported by CODE GRAY in SYSCTRL register group.

[2.1.0]

- Bug Fixes
  - Added a workaround to fix the issue that no interrupt was reported when user set smaller period.
  - Fixed violation of MISRA C-2012 rule 10.3 and 11.9.
- Improvements
  - Added return value for the two APIs to set match value.
    - OSTIMER_SetMatchRawValue
    - OSTIMER_SetMatchValue

[2.0.3]

- Bug Fixes
  - Fixed violation of MISRA C-2012 rule 10.3, 14.4, 17.7.

[2.0.2]

- Improvements
  - Added support for OSTIMER0

[2.0.1]

- Improvements
  - Removed the software reset function out of the initialization API.
  - Enabled interrupt directly instead of enabling deep sleep interrupt. Users need to enable the deep sleep interrupt in application code if needed.

[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.

PLU

[2.2.1]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.3 and rule 17.7.

[2.2.0]

- Bug Fixes
  - Fixed wrong parameter of the PLU_EnableWakeIntRequest function.

[2.1.0]

- New Features
  - Added 4 new APIs to support Niobe4's wake-up/interrupt control feature, including PLU_GetDefaultWakeIntConfig() PLU_EnableWakeIntRequest(), PLU_LatchInterrupt() and PLU_ClearLatchedInterrupt().
- Other Changes
  - Changed the register name LUT_INP to LUT_INP_MUX due to register map update.

[2.0.1]

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

[2.0.0]

- Initial version.

POWER

[2.0.1]

- Improvements
  - Use static lv_low_power_mode_cfg in POWER_EnterDeepSleep for Keil 5.30

[2.0.0]

- Initial version.

PRINCE

- Version 2.6.0
- Renamed CSS to ELS.
  - Version 2.5.1
- Fix build error due to renamed symbols.
  - Version 2.3.2
- Fix documentation of enumeration.
- Extend PRINCE example.
  - Version 2.3.1
- Fix MISRA-2012 issues.
- Add support for LPC55S0x series
  - Version 2.3.0
- Add support for LPC55S1x and LPC55S2x series
  - Version 2.2.0
- Add runtime checking of the A0 and A1 rev. of LPC55Sxx serie to support both silicone revisions.
  - Version 2.1.0
- Update for the A1 rev. of LPC55Sxx serie.

[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.

[2.0.3]

- Improvements
  - Add CASPER_RSTS,HASHCRYPT_RSTS and PUF_RSTS.

[2.0.2]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.3.

[2.0.1]

- Improvements
  - Updated component full_name to “Reset Driver”.

[2.0.0]

- Initial version.

RNG

[2.0.3]

- Modified RNG_Init and RNG_GetRandomData functions, added rng_accumulateEntropy and rng_readEntropy functions. These changes are reflecting recommended usage of RNG according to device UM

[2.0.2]

- Add RESET_PeripheralReset function inside RNG_Init and RNG_Deinit functions.

[2.0.1]

- Fix MISRA C-2012 issue.

[2.0.0]

- Initial version.

RTC

[2.2.0]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

    * RTC_DisableInterrupts
    * RTC_GetEnabledInterrupts

[2.0.0]

- Initial version.

SCTIMER

[2.5.1]

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

[2.5.0]

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

[2.4.9]

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

[2.4.8]

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

[2.4.7]

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

[2.4.6]

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

[2.4.5]

- Bug Fixes
  - Fix SCT_EV_STATE_STATEMSKn macro build error.

[2.4.4]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

[2.3.0]

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

[2.2.0]

- Improvements
  - Updated for 16-bit register access.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.

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 configuing 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.

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.

SYSCTL

[2.0.5]

- Bug Fixes:
  - Fixed violations of MISRA C-2012 rule 8.3, 10.1, 10.4, 10.7.

[2.0.4]

- Improvements:
  - Update macro name to align with the header of devices.

[2.0.3]

- Improvements:
  - Update the register and macro name to align with the header of devices.

[2.0.2]

- Removed kSYSCTL_Flexcomm3DataOut enumeration definition.

[2.0.1]

- Fixed some typo error comments and improved driver integral ability.

[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 USART_TransferSendNonBlocking.

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

- New Features
  - Added the rx timeout interrupts and status flags of bus status.
  - Added new rx timeout configuration item in usart_config_t.
  - Added API USART_SetRxTimeoutConfig for rx timeout configuration.
- Improvements
  - When the calculated baudrate cannot meet user's configuration, lower OSR value is 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 txDataSize is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.

[2.1.1]

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

[2.1.0]

- New Features
  - Added features to allow users to configure the USART to synchronous transfer(master and slave) mode.
- Bug Fixes
  - Modified USART_SetBaudRate to get more 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 FIFOINTENSET register should not be used to disable the interrupts, so use the FIFOINTENCLR register instead.

[2.0.1]

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

[2.0.0]

- Initial version.

USART_DMA

[2.6.0]

- Refer USART driver change log 2.0.1 to 2.6.0

UTICK

[2.0.5]

- Improvements
  - Improved for SOC RW610.

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.

WWDT

[2.1.10]

- 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.

LPC55S28

1.7 Middleware Documentation

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

1.7.1 FreeMASTER

freemaster

1.7.2 FreeRTOS

FreeRTOS

1.7.3 File systemFatfs

FatFs

Chapter 2

LPC55S28

2.1 ANACTRL: Analog Control Driver

void ANACTRL_Init(ANACTRL_Type *base)

Initializes the ANACTRL mode, the module's clock will be enabled by invoking this function.

Parameters

- base – ANACTRL peripheral base address.

void ANACTRL_Deinit(ANACTRL_Type *base)

De-initializes ANACTRL module, the module's clock will be disabled by invoking this function.

Parameters

- base – ANACTRL peripheral base address.

void ANACTRL_SetFro192M(ANACTRL_Type *base, const *anactrl_fro192M_config_t* *config)

Configs the on-chip high-speed Free Running Oscillator(FRO192M), such as enabling/disabling 12 MHZ clock output and enable/disable 96MHZ clock output.

Parameters

- base – ANACTRL peripheral base address.
- config – Pointer to FRO192M configuration structure. Refer to anactrl_fro192M_config_t structure.

void ANACTRL_GetDefaultFro192MConfig(*anactrl_fro192M_config_t* *config)

Gets the default configuration of FRO192M. The default values are:

```
config->enable12MHzClk = true;
config->enable96MHzClk = false;
```

Parameters

- config – Pointer to FRO192M configuration structure. Refer to anactrl_fro192M_config_t structure.

void ANACTRL_SetXo32M(ANACTRL_Type *base, const *anactrl_xo32M_config_t* *config)

Configs the 32 MHz Crystal oscillator(High-speed crystal oscillator), such as enable/disable output to CPU system, and so on.

Parameters

- base – ANACTRL peripheral base address.
- config – Pointer to XO32M configuration structure. Refer to anactrl_xo32M_config_t structure.

void ANACTRL_GetDefaultXo32MConfig(*anactrl_xo32M_config_t* *config)

Gets the default configuration of XO32M. The default values are:

```
config->enableSysCLkOutput = false;
config->enableACBufferBypass = false;
```

Parameters

- config – Pointer to XO32M configuration structure. Refer to anactrl_xo32M_config_t structure.

uint32_t ANACTRL_MeasureFrequency(ANACTRL_Type *base, uint8_t scale, uint32_t refClkFreq)

Measures the frequency of the target clock source.

This function measures target frequency according to a accurate reference frequency.The formula is: Ftarget = (CAPVAL * Freference) / ((1«SCALE)-1)

Note: Both tartget and reference clocks are selectable by programming the target clock select FREQMEAS_TARGET register in INPUTMUX and reference clock select FREQMEAS_REF register in INPUTMUX.

Parameters

- base – ANACTRL peripheral base address.
- scale – Define the power of 2 count that ref counter counts to during measurement, ranges from 2 to 31.
- refClkFreq – frequency of the reference clock.

Returns

frequency of the target clock.

static inline void ANACTRL_EnableInterrupts(ANACTRL_Type *base, uint32_t mask)

Enables the ANACTRL interrupts.

Parameters

- base – ANACTRL peripheral base address.
- mask – The interrupt mask. Refer to "_anactrl_interrupt" enumeration.

static inline void ANACTRL_DisableInterrupts(ANACTRL_Type *base, uint32_t mask)

Disables the ANACTRL interrupts.

Parameters

- base – ANACTRL peripheral base address.
- mask – The interrupt mask. Refer to "_anactrl_interrupt" enumeration.

static inline void ANACTRL_ClearInterrupts(ANACTRL_Type *base, uint32_t mask)

Clears the ANACTRL interrupts.

Parameters

- base – ANACTRL peripheral base address.
- mask – The interrupt mask. Refer to "_anactrl_interrupt" enumeration.

static inline uint32_t ANACTRL_GetStatusFlags(ANACTRL_Type *base)

Gets ANACTRL status flags.

This function gets Analog control status flags. The flags are returned as the logical OR value of the enumerators _anactrl_flags. To check for a specific status, compare the return value with enumerators in the _anactrl_flags. For example, to check whether the flash is in power down mode:

```
if (kANACTRL_FlashPowerDownFlag & ANACTRL_ANACTRL_GetStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- base – ANACTRL peripheral base address.

Returns

ANACTRL status flags which are given in the enumerators in the _anactrl_flags.

static inline uint32_t ANACTRL_GetOscStatusFlags(ANACTRL_Type *base)

Gets ANACTRL oscillators status flags.

This function gets Anactrl oscillators status flags. The flags are returned as the logical OR value of the enumerators _anactrl_osc_flags. To check for a specific status, compare the return value with enumerators in the _anactrl_osc_flags. For example, to check whether the FRO192M clock output is valid:

```
if (kANACTRL_OutputClkValidFlag & ANACTRL_ANACTRL_GetOscStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- base – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the _anactrl_osc_flags.

static inline uint32_t ANACTRL_GetInterruptStatusFlags(ANACTRL_Type *base)

Gets ANACTRL interrupt status flags.

This function gets Anactrl interrupt status flags. The flags are returned as the logical OR value of the enumerators _anactrl_interrupt_flags. To check for a specific status, compare the return value with enumerators in the _anactrl_interrupt_flags. For example, to check whether the VBAT voltage level is above the threshold:

```
if (kANACTRL_BodVbatPowerFlag & ANACTRL_ANACTRL_GetInterruptStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- base – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the _anactrl_osc_flags.

static inline void ANACTRL_EnableVref1V(ANACTRL_Type *base, bool enable)

Aux_Bias Control Interfaces.

Enables/disabless 1V reference voltage buffer.

Parameters

- base – ANACTRL peripheral base address.
- enable – Used to enable or disable 1V reference voltage buffer.

enum _anactrl_interrupt_flags

ANACTRL interrupt flags.

Values:

enumerator kANACTRL_BodVbatFlag

BOD VBAT Interrupt status before Interrupt Enable.

enumerator kANACTRL_BodVbatInterruptFlag

BOD VBAT Interrupt status after Interrupt Enable.

enumerator kANACTRL_BodVbatPowerFlag

Current value of BOD VBAT power status output.

enumerator kANACTRL_BodCoreFlag

BOD CORE Interrupt status before Interrupt Enable.

enumerator kANACTRL_BodCoreInterruptFlag

BOD CORE Interrupt status after Interrupt Enable.

enumerator kANACTRL_BodCorePowerFlag

Current value of BOD CORE power status output.

enumerator kANACTRL_DcdcFlag

DCDC Interrupt status before Interrupt Enable.

enumerator kANACTRL_DcdcInterruptFlag

DCDC Interrupt status after Interrupt Enable.

enumerator kANACTRL_DcdcPowerFlag

Current value of DCDC power status output.

enum _anactrl_interrupt

ANACTRL interrupt control.

Values:

enumerator kANACTRL_BodVbatInterruptEnable

BOD VBAT interrupt control.

enumerator kANACTRL_BodCoreInterruptEnable

BOD CORE interrupt control.

enumerator kANACTRL_DcdcInterruptEnable

DCDC interrupt control.

enum _anactrl_flags

ANACTRL status flags.

Values:

enumerator kANACTRL_FlashPowerDownFlag

Flash power-down status.

enumerator kANACTRL_FlashInitErrorFlag
Flash initialization error status.

enum _anactrl_osc_flags
ANACTRL FRO192M and XO32M status flags.

Values:

enumerator kANACTRL_OutputClkValidFlag
Output clock valid signal.

enumerator kANACTRL_CCOThresholdVoltageFlag
CCO threshold voltage detector output (signal vcco_ok).

enumerator kANACTRL_XO32MOutputReadyFlag
Indicates XO out frequency statibilty.

typedef struct *_anactrl_fro192M_config* anactrl_fro192M_config_t
Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

typedef struct *_anactrl_xo32M_config* anactrl_xo32M_config_t
Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

FSL_ANACTRL_DRIVER_VERSION
ANACTRL driver version.

struct _anactrl_fro192M_config
#include <fsl_anactrl.h> Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

Public Members

bool enable12MHzClk
Enable 12MHz clock.

bool enable96MHzClk
Enable 96MHz clock.

struct _anactrl_xo32M_config
#include <fsl_anactrl.h> Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

Public Members

bool enableACBufferBypass
: Enable XO AC buffer bypass in pll and top level.

bool enableSysCLkOutput
: Enable XO 32 MHz output to CPU system, SCT, and CLKOUT

bool enableADCOutput
: Enable High speed crystal oscillator output to ADC.

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

2.3 casper_driver

FSL_CASPER_DRIVER_VERSION
: CASPER driver version. Version 2.2.4.

Current version: 2.2.4

Change log:

- Version 2.0.0
  - Initial version
- Version 2.0.1
  - Bug fix KPSDK-24531 double_scalar_multiplication() result may be all zeroes for some specific input
- Version 2.0.2
  - Bug fix KPSDK-25015 CASPER_MEMCPY hard-fault on LPC55xx when both source and destination buffers are outside of CASPER_RAM
- Version 2.0.3
  - Bug fix KPSDK-28107 RSUB, FILL and ZERO operations not implemented in enum _casper_operation.
- Version 2.0.4
  - For GCC compiler, enforce O1 optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires -fno-strict-aliasing.
- Version 2.0.5
  - Fix sign-compare warning.
- Version 2.0.6
  - Fix IAR Pa082 warning.
- Version 2.0.7
  - Fix MISRA-C 2012 issue.
- Version 2.0.8
  - Add feature macro for CASPER_RAM_OFFSET.
- Version 2.0.9
  - Remove unused function Jac_oncurve().
  - Fix ECC384 build.

- Version 2.0.10
  - Fix MISRA-C 2012 issue.
- Version 2.1.0
  - Add ECC NIST P-521 elliptic curve.
- Version 2.2.0
  - Rework driver to support multiple curves at once.
- Version 2.2.1
  - Fix MISRA-C 2012 issue.
- Version 2.2.2
  - Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE
- Version 2.2.3
  - Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.
- Version 2.2.4
  - Fix MISRA-C 2012 issue.

enum _casper_operation

CASPER operation.

Values:

enumerator kCASPER_OpMul6464NoSum

enumerator kCASPER_OpMul6464Sum

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

enumerator kCASPER_OpMul6464FullSum

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

enumerator kCASPER_OpMul6464Reduce

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

enumerator kCASPER_OpAdd64

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

enumerator kCASPER_OpSub64

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

enumerator kCASPER_OpDouble64

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

enumerator kCASPER_OpXor64

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

enumerator kCASPER_OpRSub64

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

enumerator kCASPER_OpShiftLeft32

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

enumerator kCASPER_OpShiftRight32

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

enumerator kCASPER_OpCopy

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

enumerator kCASPER_OpRemask

Copy from ABoff to resoff, 64b at a time

enumerator kCASPER_OpFill

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

enumerator kCASPER_OpZero

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

enumerator kCASPER_OpCompare

Fill RESOFF using 64 bits at a time of 0s

enumerator kCASPER_OpCompareFast

Compare two arrays, running all the way to the end

enum _casper_algo_t

Algorithm used for CASPER operation.

Values:

enumerator kCASPER_ECC_P256

ECC_P256

enumerator kCASPER_ECC_P384

ECC_P384

enumerator kCASPER_ECC_P521

ECC_P521

Values:

enumerator kCASPER_RamOffset_Result

enumerator kCASPER_RamOffset_Base

enumerator kCASPER_RamOffset_TempBase

enumerator kCASPER_RamOffset_Modulus

enumerator kCASPER_RamOffset_M64

typedef enum *_casper_operation* casper_operation_t

CASPER operation.

typedef enum *_casper_algo_t* casper_algo_t

Algorithm used for CASPER operation.

void CASPER_Init(CASPER_Type *base)

Enables clock and disables reset for CASPER peripheral.

Enable clock and disable reset for CASPER.

Parameters

- base – CASPER base address

void CASPER_Deinit(CASPER_Type *base)

Disables clock for CASPER peripheral.

Disable clock and enable reset.

Parameters

- base – CASPER base address

CASPER_CP

CASPER_CP_CTRL0

CASPER_CP_CTRL1

CASPER_CP_LOADER

CASPER_CP_STATUS

CASPER_CP_INTENSET

CASPER_CP_INTENCLR

CASPER_CP_INTSTAT

CASPER_CP_AREG

CASPER_CP_BREG

CASPER_CP_CREG

CASPER_CP_DREG

CASPER_CP_RES0

CASPER_CP_RES1

CASPER_CP_RES2

CASPER_CP_RES3

CASPER_CP_MASK

CASPER_CP_REMASK

CASPER_CP_LOCK

CASPER_CP_ID

CASPER_Wr32b(value, off)

CASPER_Wr64b(value, off)

CASPER_Rd32b(off)

N_wordlen_max

2.4 casper_driver_pkha

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

Performs modular exponentiation - (A^E) mod N.

This function performs modular exponentiation.

Parameters

- base – CASPER base address

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

void CASPER_ecc_init(*casper_algo_t* curve)

Initialize prime modulus mod in Casper memory .

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

Parameters

- curve – elliptic curve algoritm

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

Performs ECC secp256r1 point single scalar multiplication.

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

Parameters

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

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

Performs ECC secp256r1 point double scalar multiplication.

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

Parameters

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

- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

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

Performs ECC secp384r1 point single scalar multiplication.

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

Parameters

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

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

Performs ECC secp384r1 point double scalar multiplication.

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

Parameters

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

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

Performs ECC secp521r1 point single scalar multiplication.

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

Parameters

- base – CASPER base address

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

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

Performs ECC secp521r1 point double scalar multiplication.

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

Parameters

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

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

void CASPER_ECC_equal_to_zero(int *res, uint32_t *op1)

2.5 CMP: Analog Comparator Driver

void CMP_Init(const *cmp_config_t* *config)

CMP initialization.

This function enables the CMP module and do necessary settings.

Parameters

- config – Pointer to the configuration structure.

void CMP_Deinit(void)

CMP deinitialization.

This function gates the clock for CMP module.

void CMP_GetDefaultConfig(*cmp_config_t* *config)

Initializes the CMP user configuration structure.

This function initializes the user configuration structure to these default values.

```
config->enableHysteresis    = true;
config->enableLowPower      = true;
config->filterClockDivider  = kCMP_FilterClockDivide1;
config->filterSampleMode    = kCMP_FilterSampleMode0;
```

Parameters

- config – Pointer to the configuration structure.

static inline void CMP_SetInputChannels(uint8_t positiveChannel, uint8_t negativeChannel)

void CMP_SetVREF(const *cmp_vref_config_t* *config)

Configures the VREFINPUT.

Parameters

- config – Pointer to the configuration structure.

static inline bool CMP_GetOutput(void)

Get CMP compare output.

Returns

The output result. true: voltage on positive side is greater than negative side. false: voltage on positive side is lower than negative side.

static inline void CMP_EnableInterrupt(uint32_t type)

CMP enable interrupt.

Parameters

- type – CMP interrupt type. See "_cmp_interrupt_type".

static inline void CMP_DisableInterrupt(void)

CMP disable interrupt.

static inline void CMP_ClearInterrupt(void)

CMP clear interrupt.

static inline void CMP_EnableFilteredInterruptSource(bool enable)

Select which Analog comparator output (filtered or un-filtered) is used for interrupt detection.

Note: : When CMP is configured as the wakeup source in power down mode, this function must use the raw output as the interupt source, that is, call this function and set parameter enable to false.

Parameters

- enable – false: Select Analog Comparator raw output (unfiltered) as input for interrupt detection. true: Select Analog Comparator filtered output as input for interrupt detection.

static inline bool CMP_GetPreviousInterruptStatus(void)

Get CMP interrupt status before interupt enable.

Returns

Interrupt status. true: interrupt pending, false: no interrupt pending.

static inline bool CMP_GetInterruptStatus(void)

Get CMP interrupt status after interupt enable.

Returns

Interrupt status. true: interrupt pending, false: no interrupt pending.

static inline void CMP_FilterSampleConfig(*cmp_filtercgf_samplemode_t* filterSampleMode, *cmp_filtercgf_clkdiv_t* filterClockDivider)

CMP Filter Sample Config.

This function allows the users to configure the sampling mode and clock divider of the CMP Filter.

Parameters

- filterSampleMode – CMP Select filter sample mode
- filterClockDivider – CMP Set fileter clock divider

FSL_CMP_DRIVER_VERSION

Driver version 2.2.1.

enum _cmp_input_mux

CMP input mux for positive and negative sides.

Values:

enumerator kCMP_InputVREF

Cmp input from VREF.

enumerator kCMP_Input1

Cmp input source 1.

enumerator kCMP_Input2

Cmp input source 2.

enumerator kCMP_Input3

Cmp input source 3.

enumerator kCMP_Input4

Cmp input source 4.

enumerator kCMP_Input5

Cmp input source 5.

enum _cmp_interrupt_type

CMP interrupt type.

Values:

enumerator kCMP_EdgeDisable

Disable edge interupt.

enumerator kCMP_EdgeRising

Interrupt on falling edge.

enumerator kCMP_EdgeFalling

Interrupt on rising edge.

enumerator kCMP_EdgeRisingFalling

Interrupt on both rising and falling edges.

enumerator kCMP_LevelDisable

Disable level interupt.

enumerator kCMP_LevelHigh

Interrupt on high level.

enumerator kCMP_LevelLow

Interrupt on low level.

enum _cmp_vref_source
CMP Voltage Reference source.

Values:

enumerator kCMP_VREFSourceVDDA
Select VDDA as VREF.

enumerator kCMP_VREFSourceInternalVREF
Select internal VREF as VREF.

enum _cmp_filtercgf_samplemode
CMP Filter sample mode.

Values:

enumerator kCMP_FilterSampleMode0
Bypass mode. Filtering is disabled.

enumerator kCMP_FilterSampleMode1
Filter 1 clock period.

enumerator kCMP_FilterSampleMode2
Filter 2 clock period.

enumerator kCMP_FilterSampleMode3
Filter 3 clock period.

enum _cmp_filtercgf_clkdiv
CMP Filter clock divider.

Values:

enumerator kCMP_FilterClockDivide1
Filter clock period duration equals 1 analog comparator clock period.

enumerator kCMP_FilterClockDivide2
Filter clock period duration equals 2 analog comparator clock period.

enumerator kCMP_FilterClockDivide4
Filter clock period duration equals 4 analog comparator clock period.

enumerator kCMP_FilterClockDivide8
Filter clock period duration equals 8 analog comparator clock period.

enumerator kCMP_FilterClockDivide16
Filter clock period duration equals 16 analog comparator clock period.

enumerator kCMP_FilterClockDivide32
Filter clock period duration equals 32 analog comparator clock period.

enumerator kCMP_FilterClockDivide64
Filter clock period duration equals 64 analog comparator clock period.

typedef enum *_cmp_vref_source* cmp_vref_source_t
CMP Voltage Reference source.

typedef struct *_cmp_vref_config* cmp_vref_config_t

typedef enum *_cmp_filtercgf_samplemode* cmp_filtercgf_samplemode_t
CMP Filter sample mode.

typedef enum *_cmp_filtercgf_clkdiv* cmp_filtercgf_clkdiv_t
CMP Filter clock divider.

typedef struct *_cmp_config* cmp_config_t
CMP configuration structure.

struct _cmp_vref_config
#include <fsl_cmp.h>

Public Members

cmp_vref_source_t vrefSource
Reference voltage source.

uint8_t vrefValue
Reference voltage step. Available range is 0-31. Per step equals to VREFINPUT/31.

struct _cmp_config
#include <fsl_cmp.h> CMP configuration structure.

Public Members

bool enableHysteresis
Enable hysteresis.

bool enableLowPower
Enable low power mode.

2.6 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.

2.7 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

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

2.8 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.

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),
```

(continues on next page)

(continued from previous page)

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

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

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

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

Parameters

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

Return values

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

void DMA_StartTransfer(*dma_handle_t* *handle)

DMA start transfer.

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

Parameters

- handle – DMA handle pointer.

void DMA_IRQHandle(DMA_Type *base)

DMA IRQ handler for descriptor transfer complete.

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

Parameters

- base – DMA base address.

FSL_DMA_DRIVER_VERSION

DMA driver version.

Version 2.5.4.

_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

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

- 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

2.9 IAP: In Application Programming Driver

enum __flash__driver__version__constants
Flash driver version for ROM.

Values:

enumerator kFLASH__DriverVersionName
Flash driver version name.

enumerator kFLASH__DriverVersionMajor
Major flash driver version.

enumerator kFLASH__DriverVersionMinor
Minor flash driver version.

enumerator kFLASH__DriverVersionBugfix
Bugfix for flash driver version.

MAKE__VERSION(major, minor, bugfix)
Constructs the version number for drivers.

FSL__FLASH__DRIVER__VERSION
Flash driver version for SDK.

Version 2.1.5.

enum _flash_status

Flash driver status codes.

Values:

enumerator kStatus_FLASH_Success
API is executed successfully

enumerator kStatus_FLASH_InvalidArgument
Invalid argument

enumerator kStatus_FLASH_SizeError
Error size

enumerator kStatus_FLASH_AlignmentError
Parameter is not aligned with the specified baseline

enumerator kStatus_FLASH_AddressError
Address is out of range

enumerator kStatus_FLASH_AccessError
Invalid instruction codes and out-of bound addresses

enumerator kStatus_FLASH_ProtectionViolation
The program/erase operation is requested to execute on protected areas

enumerator kStatus_FLASH_CommandFailure
Run-time error during command execution.

enumerator kStatus_FLASH_UnknownProperty
Unknown property.

enumerator kStatus_FLASH_EraseKeyError
API erase key is invalid.

enumerator kStatus_FLASH_RegionExecuteOnly
The current region is execute-only.

enumerator kStatus_FLASH_ExecuteInRamFunctionNotReady
Execute-in-RAM function is not available.

enumerator kStatus_FLASH_CommandNotSupported
Flash API is not supported.

enumerator kStatus_FLASH_ReadOnlyProperty
The flash property is read-only.

enumerator kStatus_FLASH_InvalidPropertyValue
The flash property value is out of range.

enumerator kStatus_FLASH_InvalidSpeculationOption
The option of flash prefetch speculation is invalid.

enumerator kStatus_FLASH_EccError
A correctable or uncorrectable error during command execution.

enumerator kStatus_FLASH_CompareError
Destination and source memory contents do not match.

enumerator kStatus_FLASH_RegulationLoss
A loss of regulation during read.

enumerator kStatus_FLASH_InvalidWaitStateCycles
The wait state cycle set to r/w mode is invalid.

enumerator kStatus_FLASH_OutOfDateCfpaPage
CFPA page version is out of date.

enumerator kStatus_FLASH_BlankIfrPageData
Blank page cannnot be read.

enumerator kStatus_FLASH_EncryptedRegionsEraseNotDoneAtOnce
Encrypted flash subregions are not erased at once.

enumerator kStatus_FLASH_ProgramVerificationNotAllowed
Program verification is not allowed when the encryption is enabled.

enumerator kStatus_FLASH_HashCheckError
Hash check of page data is failed.

enumerator kStatus_FLASH_SealedFfrRegion
The FFR region is sealed.

enumerator kStatus_FLASH_FfrRegionWriteBroken
The FFR Spec region is not allowed to be written discontinuously.

enumerator kStatus_FLASH_NmpaAccessNotAllowed
The NMPA region is not allowed to be read/written/erased.

enumerator kStatus_FLASH_CmpaCfgDirectEraseNotAllowed
The CMPA Cfg region is not allowed to be erased directly.

enumerator kStatus_FLASH_FfrBankIsLocked
The FFR bank region is locked.

enumerator kStatus_FLASH_EraseFrequencyError
Core frequency is over 100MHZ.

enumerator kStatus_FLASH_ProgramFrequencyError
Core frequency is over 100MHZ.

kStatusGroupGeneric
Flash driver status group.

kStatusGroupFlashDriver

MAKE_STATUS(group, code)
Constructs a status code value from a group and a code number.

enum _flash_driver_api_keys
Enumeration for Flash driver API keys.

Note: The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Values:

enumerator kFLASH_ApiEraseKey
Key value used to validate all flash erase APIs.

FOUR_CHAR_CODE(a, b, c, d)
Constructs the four character code for the Flash driver API key.

status_t FLASH_Init(*flash_config_t* *config)

Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

status_t FLASH_Erase(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)

Erases the flash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

- config – The pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be erased. The start address need to be 512bytes-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words) to be erased. Must be 512bytes-aligned.
- key – The value used to validate all flash erase APIs.

Return values

- kStatus_FLASH_Success – API was executed successfully; the appropriate number of flash sectors based on the desired start address and length were erased successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – The parameter is not aligned with the specified baseline.
- kStatus_FLASH_AddressError – The address is out of range.
- kStatus_FLASH_EraseKeyError – The API erase key is invalid.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

status_t FLASH_Program(*flash_config_t* *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be programmed. Must be 512bytes-aligned.
- src – A pointer to the source buffer of data that is to be programmed into the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be 512bytes-aligned.

Return values

- kStatus_FLASH_Success – API was executed successfully; the desired data were programed successfully into flash based on desired start address and length.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – Parameter is not aligned with the specified baseline.
- kStatus_FLASH_AddressError – Address is out of range.
- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

status_t FLASH_VerifyErase(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes)

Verifies an erasure of the desired flash area at a specified margin level.

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be verified. The start address need to be 512bytes-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. Must be 512bytes-aligned.

Return values

- kStatus_FLASH_Success – API was executed successfully; the specified FLASH region has been erased.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – Parameter is not aligned with specified baseline.
- kStatus_FLASH_AddressError – Address is out of range.
- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

status_t FLASH_VerifyProgram(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, uint32_t *failedAddress, uint32_t *failedData)

Verifies programming of the desired flash area at a specified margin level.

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be verified. need be 512bytes-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. need be 512bytes-aligned.
- expectedData – A pointer to the expected data that is to be verified against.
- failedAddress – A pointer to the returned failing address.
- failedData – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- kStatus_FLASH_Success – API was executed successfully; the desired data have been successfully programed into specified FLASH region.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – Parameter is not aligned with specified baseline.
- kStatus_FLASH_AddressError – Address is out of range.
- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

status_t FLASH_GetProperty(*flash_config_t* *config, *flash_property_tag_t* whichProperty, uint32_t *value)

Returns the desired flash property.

Parameters

- config – A pointer to the storage for the driver runtime state.
- whichProperty – The desired property from the list of properties in enum flash_property_tag_t
- value – A pointer to the value returned for the desired flash property.

Return values

- kStatus_FLASH_Success – API was executed successfully; the flash property was stored to value.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_UnknownProperty – An unknown property tag.

void BOOTLOADER_UserEntry(void *arg)

Run the Bootloader API to force into the ISP mode base on the user arg.

Parameters

- arg – Indicates API prototype fields definition. Refer to the above user_app_boot_invoke_option_t structure

FSL_FEATURE_FLASH_IP_IS_C040HD_ATFC

Flash IP Type.

FSL_FEATURE_FLASH_IP_IS_C040HD_FC

enum _flash_property_tag

Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflashSectorSize

Pflash sector size property.

enumerator kFLASH_PropertyPflashTotalSize

Pflash total size property.

enumerator kFLASH_PropertyPflashBlockSize

Pflash block size property.

enumerator kFLASH_PropertyPflashBlockCount

Pflash block count property.

enumerator kFLASH_PropertyPflashBlockBaseAddr

Pflash block base address property.

enumerator kFLASH_PropertyPflashPageSize

Pflash page size property.

enumerator kFLASH_PropertyPflashSystemFreq

System Frequency System Frequency.

enumerator kFLASH_PropertyFfrSectorSize

FFR sector size property.

enumerator kFLASH_PropertyFfrTotalSize

FFR total size property.

enumerator kFLASH_PropertyFfrBlockBaseAddr

FFR block base address property.

enumerator kFLASH_PropertyFfrPageSize

FFR page size property.

enum _flash_max_erase_page_value

Enumeration for flash max pages to erase.

Values:

enumerator kFLASH_MaxPagesToErase

The max value in pages to erase.

enum _flash_alignment_property

Enumeration for flash alignment property.

Values:

enumerator kFLASH_AlignementUnitVerifyErase

The alignment unit in bytes used for verify erase operation.

enumerator kFLASH_AlignementUnitProgram

The alignment unit in bytes used for program operation.

enumerator kFLASH_AlignementUnitSingleWordRead

The alignment unit in bytes used for verify program operation. The alignment unit in bytes used for SingleWordRead command.

enum _flash_read_ecc_option

Enumeration for flash read ecc option.

Values:

enumerator kFLASH_ReadWithEccOn

enumerator kFLASH_ReadWithEccOff

ECC is on

enum _flash_freq_tag

Values:

enumerator kSysToFlashFreq_lowInMHz

enumerator kSysToFlashFreq_defaultInMHz

enum _flash_read_margin_option

Enumeration for flash read margin option.

Values:

enumerator kFLASH_ReadMarginNormal

Normal read

enumerator kFLASH_ReadMarginVsProgram

Margin vs. program

enumerator kFLASH_ReadMarginVsErase

Margin vs. erase

enumerator kFLASH_ReadMarginIllegalBitCombination

Illegal bit combination

enum _flash_read_dmacc_option

Enumeration for flash read dmacc option.

Values:

enumerator kFLASH_ReadDmaccDisabled

Memory word

enumerator kFLASH_ReadDmaccEnabled
DMACC word

enum _flash_ramp_control_option
Enumeration for flash ramp control option.

Values:

enumerator kFLASH_RampControlDivisionFactorReserved
Reserved

enumerator kFLASH_RampControlDivisionFactor256
clk48mhz / 256 = 187.5KHz

enumerator kFLASH_RampControlDivisionFactor128
clk48mhz / 128 = 375KHz

enumerator kFLASH_RampControlDivisionFactor64
clk48mhz / 64 = 750KHz

typedef enum *_flash_property_tag* flash_property_tag_t
Enumeration for various flash properties.

typedef struct *_flash_ecc_log* flash_ecc_log_t
Flash ECC log info.

typedef struct *_flash_mode_config* flash_mode_config_t
Flash controller paramter config.

typedef struct *_flash_ffr_config* flash_ffr_config_t
Flash controller paramter config.

typedef struct *_flash_config* flash_config_t
Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

status_t FLASH_Read(*flash_config_t* *config, uint32_t start, uint8_t *dest, uint32_t lengthInBytes)
Reads flash at locations passed in through parameters.

This function read the flash memory from a given flash area as determined by the start address and the length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be read.
- dest – A pointer to the dest buffer of data that is to be read from the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be read.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – Parameter is not aligned with the specified baseline.
- kStatus_FLASH_AddressError – Address is out of range.

- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

struct _flash_ecc_log

#include <fsl_iap.h> Flash ECC log info.

struct _flash_mode_config

#include <fsl_iap.h> Flash controller paramter config.

struct _flash_ffr_config

#include <fsl_iap.h> Flash controller paramter config.

struct _flash_config

#include <fsl_iap.h> Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

Public Members

uint32_t PFlashBlockBase

A base address of the first PFlash block

uint32_t PFlashTotalSize

The size of the combined PFlash block.

uint32_t PFlashBlockCount

A number of PFlash blocks.

uint32_t PFlashPageSize

The size in bytes of a page of PFlash.

uint32_t PFlashSectorSize

The size in bytes of a sector of PFlash.

struct user_app_boot_invoke_option_t

#include <fsl_iap.h>

struct readSingleWord

struct setWriteMode

struct setReadMode

union option

Public Members

struct *user_app_boot_invoke_option_t* B

uint32_t U

struct B

2.10 IAP_FFR Driver

status_t FFR_Init(*flash_config_t* *config)

Initializes the global FFR properties structure members.

Parameters

- config – A pointer to the storage for the driver runtime state.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.

status_t FFR_Lock_All(*flash_config_t* *config)

Enable firewall for all flash banks.

CFPA, CMPA, and NMPA flash areas region will be locked, After this function executed; Unless the board is reset again.

Parameters

- config – A pointer to the storage for the driver runtime state.

Return values

- kStatus_FLASH_Success – An invalid argument is provided.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.

status_t FFR_InfieldPageWrite(*flash_config_t* *config, uint8_t *page_data, uint32_t valid_len)

APIs to access CFPA pages.

This routine will erase CFPA and program the CFPA page with passed data.

Parameters

- config – A pointer to the storage for the driver runtime state.
- page_data – A pointer to the source buffer of data that is to be programmed into the CFPA.
- valid_len – The length, given in bytes, to be programmed.

Return values

- kStatus_FLASH_Success – The desire page-data were programed successfully into CFPA.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FLASH_FfrBankIsLocked – The CFPA was locked.
- kStatus_FLASH_OutOfDateCfpaPage – It is not newest CFPA page.

status_t FFR_GetCustomerInfieldData(*flash_config_t* *config, uint8_t *pData, uint32_t offset, uint32_t len)

APIs to access CFPA pages.

Generic read function, used by customer to read data stored in 'Customer In-field Page'.

Parameters

- config – A pointer to the storage for the driver runtime state.
- pData – A pointer to the dest buffer of data that is to be read from 'Customer In-field Page'.

- offset – An offset from the 'Customer In-field Page' start address.
- len – The length, given in bytes, to be read.

Return values

- kStatus_FLASH_Success – Get data from 'Customer In-field Page'.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FLASH_CommandFailure – access error.

status_t FFR_CustFactoryPageWrite(*flash_config_t* *config, uint8_t *page_data, bool seal_part)

APIs to access CMPA pages.

This routine will erase "customer factory page" and program the page with passed data. If 'seal_part' parameter is TRUE then the routine will compute SHA256 hash of the page contents and then programs the pages. 1.During development customer code uses this API with 'seal_part' set to FALSE. 2.During manufacturing this parameter should be set to TRUE to seal the part from further modifications 3.This routine checks if the page is sealed or not. A page is said to be sealed if the SHA256 value in the page has non-zero value. On boot ROM locks the firewall for the region if hash is programmed anyways. So, write/erase commands will fail eventually.

Parameters

- config – A pointer to the storage for the driver runtime state.
- page_data – A pointer to the source buffer of data that is to be programmed into the "customer factory page".
- seal_part – Set fasle for During development customer code.

Return values

- kStatus_FLASH_Success – The desire page-data were programed successfully into CMPA.
- kStatus_FLASH_InvalidArgument – Parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FLASH_CommandFailure – access error.

status_t FFR_GetCustomerData(*flash_config_t* *config, uint8_t *pData, uint32_t offset, uint32_t len)

APIs to access CMPA page.

Read data stored in 'Customer Factory CFG Page'.

Parameters

- config – A pointer to the storage for the driver runtime state.
- pData – A pointer to the dest buffer of data that is to be read from the Customer Factory CFG Page.
- offset – Address offset relative to the CMPA area.
- len – The length, given in bytes to be read.

Return values

- kStatus_FLASH_Success – Get data from 'Customer Factory CFG Page'.
- kStatus_FLASH_InvalidArgument – Parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FLASH_CommandFailure – access error.

status_t FFR_GetUUID(*flash_config_t* *config, uint8_t *uuid)

APIs to access CMPA page.

1.SW should use this API routine to get the UUID of the chip. 2.Calling routine should pass a pointer to buffer which can hold 128-bit value.

status_t FFR_KeystoreWrite(*flash_config_t* *config, *ffr_key_store_t* *pKeyStore)

This routine writes the 3 pages allocated for Key store data,.

1.Used during manufacturing. Should write pages when 'customer factory page' is not in sealed state. 2.Optional routines to set individual data members (activation code, key codes etc) to construct the key store structure in RAM before committing it to IFR/FFR.

Parameters

- config – A pointer to the storage for the driver runtime state.
- pKeyStore – A Pointer to the 3 pages allocated for Key store data. that will be written to 'customer factory page'.

Return values

- kStatus_FLASH_Success – The key were programed successfully into FFR.
- kStatus_FLASH_InvalidArgument – Parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FLASH_CommandFailure – access error.

status_t FFR_KeystoreGetAC(*flash_config_t* *config, uint8_t *pActivationCode)

Get/Read Key store code routines.

a. Calling code should pass buffer pointer which can hold activation code 1192 bytes.
b. Check if flash aperture is small or regular and read the data appropriately.

status_t FFR_KeystoreGetKC(*flash_config_t* *config, uint8_t *pKeyCode, *ffr_key_type_t* keyIndex)

Get/Read Key store code routines.

a. Calling code should pass buffer pointer which can hold key code 52 bytes.
b. Check if flash aperture is small or regular and read the data appropriately.
c. keyIndex specifies which key code is read.

FSL_FLASH_IFR_DRIVER_VERSION

Flash IFR driver version for SDK.

Version 2.1.0.

enum _flash_ffr_page_offset

flash ffr page offset.

Values:

enumerator kFfrPageOffset_CFPA

Customer In-Field programmed area

enumerator kFfrPageOffset_CFPA_Scratch

CFPA Scratch page

enumerator kFfrPageOffset_CFPA_Cfg
CFPA Configuration area (Ping page)

enumerator kFfrPageOffset_CFPA_CfgPong
Same as CFPA page (Pong page)

enumerator kFfrPageOffset_CMPA
Customer Manufacturing programmed area

enumerator kFfrPageOffset_CMPA_Cfg
CMPA Configuration area (Part of CMPA)

enumerator kFfrPageOffset_CMPA_Key
Key Store area (Part of CMPA)

enumerator kFfrPageOffset_NMPA
NXP Manufacturing programmed area

enumerator kFfrPageOffset_NMPA_Romcp
ROM patch area (Part of NMPA)

enumerator kFfrPageOffset_NMPA_Repair
Repair area (Part of NMPA)

enumerator kFfrPageOffset_NMPA_Cfg
NMPA configuration area (Part of NMPA)

enumerator kFfrPageOffset_NMPA_End
Reserved (Part of NMPA)

enum _flash_ffr_page_num
flash ffr page number.

Values:

enumerator kFfrPageNum_CFPA
Customer In-Field programmed area

enumerator kFfrPageNum_CMPA
Customer Manufacturing programmed area

enumerator kFfrPageNum_NMPA
NXP Manufacturing programmed area

enumerator kFfrPageNum_CMPA_Cfg

enumerator kFfrPageNum_CMPA_Key

enumerator kFfrPageNum_NMPA_Romcp

enumerator kFfrPageNum_SpecArea

enumerator kFfrPageNum_Total

enum _flash_ffr_block_size
Values:

enumerator kFfrBlockSize_Key

enumerator kFfrBlockSize_ActivationCode

enum _cfpa_cfg_cmpa_prog_process
Values:

enumerator kFfrCmpaProgProcess_Pre

enumerator kFfrCmpaProgProcess_Post

enum _ffr_key_type

Values:

enumerator kFFR_KeyTypeSbkek

enumerator kFFR_KeyTypeUser

enumerator kFFR_KeyTypeUds

enumerator kFFR_KeyTypePrinceRegion0

enumerator kFFR_KeyTypePrinceRegion1

enumerator kFFR_KeyTypePrinceRegion2

enum _ffr_bank_type

Values:

enumerator kFFR_BankTypeBank0_NMPA

enumerator kFFR_BankTypeBank1_CMPA

enumerator kFFR_BankTypeBank2_CFPA

typedef enum *_cfpa_cfg_cmpa_prog_process* cmpa_prog_process_t

typedef struct *_cfpa_cfg_iv_code* cfpa_cfg_iv_code_t

typedef struct *_cfpa_cfg_info* cfpa_cfg_info_t

typedef struct *_cmpa_cfg_info* cmpa_cfg_info_t

typedef struct *_cmpa_key_store_header* cmpa_key_store_header_t

typedef struct *_nmpa_cfg_info* nmpa_cfg_info_t

typedef struct *_ffr_key_store* ffr_key_store_t

typedef enum *_ffr_key_type* ffr_key_type_t

typedef enum *_ffr_bank_type* ffr_bank_type_t

ALIGN_DOWN(x, a)

Alignment(down) utility.

ALIGN_UP(x, a)

Alignment(up) utility.

FLASH_FFR_MAX_PAGE_SIZE

FLASH_FFR_HASH_DIGEST_SIZE

FLASH_FFR_IV_CODE_SIZE

FFR_BOOTCFG_BOOTSPEED_MASK

FFR_BOOTCFG_BOOTSPEED_SHIFT

FFR_BOOTCFG_BOOTSPEED_48MHZ

FFR_BOOTCFG_BOOTSPEED_96MHZ

FFR_USBID_VENDORID_MASK

FFR_USBID_VENDORID_SHIFT

FFR_USBID_PRODUCTID_MASK

FFR_USBID_PRODUCTID_SHIFT

FFR_SYSTEM_SPEED_CODE_MASK

FFR_SYSTEM_SPEED_CODE_SHIFT

FFR_SYSTEM_SPEED_CODE_FRO12MHZ_12MHZ

FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_24MHZ

FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_48MHZ

FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_96MHZ

FFR_PERIPHERALCFG_PERI_MASK

FFR_PERIPHERALCFG_PERI_SHIFT

FFR_PERIPHERALCFG_COREEN_MASK

FFR_PERIPHERALCFG_COREEN_SHIFT

struct _cfpa_cfg_iv_code
#include <fsl_iap_ffr.h>

struct _cfpa_cfg_info
#include <fsl_iap_ffr.h>

Public Members

uint32_t header
[0x000-0x003]

uint32_t version
[0x004-0x007

uint32_t secureFwVersion
[0x008-0x00b

uint32_t nsFwVersion
[0x00c-0x00f]

uint32_t imageKeyRevoke
[0x010-0x013]

uint8_t reserved0[4]
[0x014-0x017]

uint32_t rotkhRevoke
[0x018-0x01b]

uint32_t vendorUsage
[0x01c-0x01f]

uint32_t dcfgNsPin
[0x020-0x013]

uint32_t dcfgNsDflt
[0x024-0x017]

uint32_t enableFaMode
[0x028-0x02b]

uint8_t reserved1[4]
[0x02c-0x02f]

cfpa_cfg_iv_code_t ivCodePrinceRegion[3]
[0x030-0x0d7]

uint8_t reserved2[264]
[0x0d8-0x1df]

uint8_t sha256[32]
[0x1e0-0x1ff]

struct __cmpa__cfg__info
#include <fsl_iap_ffr.h>

Public Members

uint32_t bootCfg
[0x000-0x003]

uint32_t spiFlashCfg
[0x004-0x007]

struct *_cmpa_cfg_info* usbId
[0x008-0x00b]

uint32_t sdioCfg
[0x00c-0x00f]

uint32_t dcfgPin
[0x010-0x013]

uint32_t dcfgDflt
[0x014-0x017]

uint32_t dapVendorUsage
[0x018-0x01b]

uint32_t secureBootCfg
[0x01c-0x01f]

uint32_t princeBaseAddr
[0x020-0x023]

uint32_t princeSr[3]
[0x024-0x02f]

uint8_t reserved0[32]
[0x030-0x04f]

uint32_t rotkh[8]
[0x050-0x06f]

uint8_t reserved1[368]
[0x070-0x1df]

uint8_t sha256[32]
[0x1e0-0x1ff]

struct __cmpa__key__store__header
#include <fsl_iap_ffr.h>

struct __nmpa__cfg__info
#include <fsl_iap_ffr.h>

Public Members

uint16_t fro32kCfg
[0x000-0x001]

uint8_t reserved0[6]
[0x002-0x007]

uint8_t sysCfg
[0x008-0x008]

uint8_t reserved1[7]
[0x009-0x00f]

struct *_nmpa_cfg_info* GpoInitData[3]
[0x010-0x03f]

uint32_t GpoDataChecksum[4]
[0x040-0x04f]

uint32_t finalTestBatchId[4]
[0x050-0x05f]

uint32_t deviceType
[0x060-0x063]

uint32_t finalTestProgVersion
[0x064-0x067]

uint32_t finalTestDate
[0x068-0x06b]

uint32_t finalTestTime
[0x06c-0x06f]

uint32_t uuid[4]
[0x070-0x07f]

uint8_t reserved2[32]
[0x080-0x09f]

uint32_t peripheralCfg
[0x0a0-0x0a3]

uint32_t ramSizeCfg
[0x0a4-0x0a7]

uint32_t flashSizeCfg
[0x0a8-0x0ab]

uint8_t reserved3[36]
[0x0ac-0x0cf]

uint8_t fro1mCfg
[0x0d0-0x0d0]

uint8_t reserved4[15]
[0x0d1-0x0df]

uint32_t dcdc[4]
[0x0e0-0x0ef]

uint32_t bod
[0x0f0-0x0f3]

uint8_t reserved5[12]
[0x0f4-0x0ff]

uint8_t calcHashReserved[192]
[0x100-0x1bf]

uint8_t sha256[32]
[0x1c0-0x1df]

uint32_t ecidBackup[4]
[0x1e0-0x1ef]

uint32_t pageChecksum[4]
[0x1f0-0x1ff]

struct __ffr_key_store
#include <fsl_iap_ffr.h>

struct usbId

struct GpoInitData

2.11 FLEXCOMM: FLEXCOMM Driver

2.12 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.

2.13 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

typedef enum *_gint_comb* gint_comb_t
GINT combine inputs type.

typedef enum *_gint_trig* gint_trig_t
GINT trigger type.

typedef enum *_gint_port* gint_port_t

typedef void (*gint_cb_t)(void)

GINT Callback function.

void GINT_Init(GINT_Type *base)

Initialize GINT peripheral.

This function initializes the GINT peripheral and enables the clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

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

Setup GINT peripheral control parameters.

This function sets the control parameters of GINT peripheral.

Parameters

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

Return values

None. –

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

Get GINT peripheral control parameters.

This function returns the control parameters of GINT peripheral.

Parameters

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

Return values

None. –

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

Configure GINT peripheral pins.

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

Parameters

- base – Base address of the GINT peripheral.
- port – Port number.
- polarityMask – Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.

- enableMask – Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

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

Get GINT peripheral pin configuration.

This function returns the pin configuration of a given port.

Parameters

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

Return values

None. –

void GINT_EnableCallback(GINT_Type *base)

Enable callback.

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

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

void GINT_DisableCallback(GINT_Type *base)

Disable callback.

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

Parameters

- base – Base address of the peripheral.

Return values

None. –

static inline void GINT_ClrStatus(GINT_Type *base)

Clear GINT status.

This function clears the GINT status bit.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

static inline uint32_t GINT_GetStatus(GINT_Type *base)

Get GINT status.

This function returns the GINT status.

Parameters

- base – Base address of the GINT peripheral.

Return values

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

void GINT_Deinit(GINT_Type *base)

Deinitialize GINT peripheral.

This function disables the GINT clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

2.14 Hashcrypt: The Cryptographic Accelerator

2.15 Hashcrypt Background HASH

void HASHCRYPT_SHA_SetCallback(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, *hashcrypt_callback_t* callback, void *userData)

Initializes the HASHCRYPT 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 – HASHCRYPT 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 HASHCRYPT_SHA_UpdateNonBlocking(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, const uint8_t *input, size_t inputSize)

Create running hash on given data.

Configures the HASHCRYPT to compute new running hash as AHB master and returns immediately. HASHCRYPT 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 HASHCRYPT_SHA_Init() and finished with HASHCRYPT_SHA_Finish(). Once callback function is invoked by HASHCRYPT isr, it should set a flag for the main application to finalize the hashing (padding) and to read out the final digest by calling HASHCRYPT_SHA_Finish().

Parameters

- base – HASHCRYPT 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.

2.16 Hashcrypt common functions

FSL_HASHCRYPT_DRIVER_VERSION

HASHCRYPT driver version. Version 2.2.16.

Current version: 2.2.16

Change log:

- Version 2.0.0
  - Initial version
- Version 2.0.1
  - Support loading AES key from unaligned address
- Version 2.0.2
  - Support loading AES key from unaligned address for different compiler and core variants
- Version 2.0.3
  - Remove SHA512 and AES ICB algorithm definitions
- Version 2.0.4
  - Add SHA context switch support
- Version 2.1.0
  - Update the register name and macro to align with new header.
- Version 2.1.1
  - Fix MISRA C-2012.
- Version 2.1.2
  - Support loading AES input data from unaligned address.
- Version 2.1.3
  - Fix MISRA C-2012.
- Version 2.1.4
  - Fix context switch cannot work when switching from AES.
- Version 2.1.5
  - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to prevent possible optimization issue.
- Version 2.2.0
  - Add AES-OFB and AES-CFB mixed IP/SW modes.

- Version 2.2.1
  - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() prevent compiler from reordering memory write when -O2 or higher is used.
- Version 2.2.2
  - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to fix optimization issue
- Version 2.2.3
  - Added check for size in hashcrypt_aes_one_block to prevent overflowing COUNT field in MEMCTRL register, if its bigger than COUNT field do a multiple runs.
- Version 2.2.4
  - In all HASHCRYPT_AES_xx functions have been added setting CTRL_MODE bitfield to 0 after processing data, which decreases power consumption.
- Version 2.2.5
  - Add data synchronization barrier and instruction synchronization barrier inside hashcrypt_sha_process_message_data() to fix optimization issue
- Version 2.2.6
  - Add data synchronization barrier inside HASHCRYPT_SHA_Update() and hashcrypt_get_data() function to fix optimization issue on MDK and ARMGCC release targets
- Version 2.2.7
  - Add data synchronization barrier inside HASHCRYPT_SHA_Update() to fix optimization issue on MCUX IDE release target
- Version 2.2.8
  - Unify hashcrypt hashing behavior between aligned and unaligned input data
- Version 2.2.9
  - Add handling of set ERROR bit in the STATUS register
- Version 2.2.10
  - Fix missing error statement in hashcrypt_save_running_hash()
- Version 2.2.11
  - Fix incorrect SHA-256 calculation for long messages with reload
- Version 2.2.12
  - Fix hardfault issue on the Keil compiler due to unaligned memcpy() input on some optimization levels
- Version 2.2.13
  - Added function hashcrypt_seed_prng() which loading random number into PRNG_SEED register before AES operation for SCA protection
- Version 2.2.14
  - Modify function hashcrypt_get_data() to prevent issue with unaligned access
- Version 2.2.15
  - Add wait on DIGEST BIT inside hashcrypt_sha_one_block() to fix issues with some optimization flags
- Version 2.2.16

- Add DSB instruction inside hashcrypt_sha_ldm_stm_16_words() to fix issues with some optimization flags

- Version 2.2.17
  - Fix context size when hashcrypt built with reload feature

enum _hashcrypt_algo_t

Algorithm used for Hashcrypt operation.

Values:

enumerator kHASHCRYPT_Sha1

SHA_1

enumerator kHASHCRYPT_Sha256

SHA_256

enumerator kHASHCRYPT_Aes

AES

typedef enum *_hashcrypt_algo_t* hashcrypt_algo_t

Algorithm used for Hashcrypt operation.

void HASHCRYPT_Init(HASHCRYPT_Type *base)

Enables clock and disables reset for HASHCRYPT peripheral.

Enable clock and disable reset for HASHCRYPT.

Parameters

- base – HASHCRYPT base address

void HASHCRYPT_Deinit(HASHCRYPT_Type *base)

Disables clock for HASHCRYPT peripheral.

Disable clock and enable reset.

Parameters

- base – HASHCRYPT base address

HASHCRYPT_MODE_SHA1

Algorithm definitions correspond with the values for Mode field in Control register !

HASHCRYPT_MODE_SHA256

HASHCRYPT_MODE_AES

2.17 Hashcrypt AES

enum _hashcrypt_aes_mode_t

AES mode.

Values:

enumerator kHASHCRYPT_AesEcb

AES ECB mode

enumerator kHASHCRYPT_AesCbc

AES CBC mode

enumerator kHASHCRYPT_AesCtr

AES CTR mode

enum _hashcrypt_aes_keysize_t

Size of AES key.

Values:

enumerator kHASHCRYPT_Aes128

AES 128 bit key

enumerator kHASHCRYPT_Aes192

AES 192 bit key

enumerator kHASHCRYPT_Aes256

AES 256 bit key

enumerator kHASHCRYPT_InvalidKey

AES invalid key

enum _hashcrypt_key

HASHCRYPT key source selection.

Values:

enumerator kHASHCRYPT_UserKey

HASHCRYPT user key

enumerator kHASHCRYPT_SecretKey

HASHCRYPT secret key (dedicated hw bus from PUF)

typedef enum *_hashcrypt_aes_mode_t* hashcrypt_aes_mode_t

AES mode.

typedef enum *_hashcrypt_aes_keysize_t* hashcrypt_aes_keysize_t

Size of AES key.

typedef enum *_hashcrypt_key* hashcrypt_key_t

HASHCRYPT key source selection.

typedef struct *_hashcrypt_handle* hashcrypt_handle_t

struct _hashcrypt_handle ___attribute___ ((aligned))

status_t HASHCRYPT_AES_SetKey(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *key, size_t keySize)

Set AES key to hashcrypt_handle_t struct and optionally to HASHCRYPT.

Sets the AES key for encryption/decryption with the hashcrypt_handle_t structure. The hashcrypt_handle_t input argument specifies key source.

Parameters

- base – HASHCRYPT peripheral base address.
- handle – Handle used for the request.
- key – 0-mod-4 aligned pointer to AES key.
- keySize – AES key size in bytes. Shall equal 16, 24 or 32.

Returns

status from set key operation

status_t HASHCRYPT_AES_EncryptEcb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *plaintext, uint8_t *ciphertext, size_t size)

Encrypts AES on one or multiple 128-bit block(s).

Encrypts AES. The source plaintext and destination ciphertext can overlap in system memory.

Parameters

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

Returns

Status from encrypt operation

status_t HASHCRYPT_AES_DecryptEcb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *ciphertext, uint8_t *plaintext, size_t size)

Decrypts AES on one or multiple 128-bit block(s).

Decrypts AES. The source ciphertext and destination plaintext can overlap in system memory.

Parameters

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

Returns

Status from decrypt operation

status_t HASHCRYPT_AES_EncryptCbc(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])

Encrypts AES using CBC block mode.

Parameters

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

Returns

Status from encrypt operation

status_t HASHCRYPT_AES_DecryptCbc(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])

Decrypts AES using CBC block mode.

Parameters

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

Returns

Status from decrypt operation

status_t HASHCRYPT_AES_CryptCtr(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *input, uint8_t *output, size_t size, uint8_t counter[16U], uint8_t counterlast[16U], 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 – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- 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 (statefull encryption). 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 encrypt operation

status_t HASHCRYPT_AES_CryptOfb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *input, uint8_t *output, size_t size, const uint8_t iv[16U])

Encrypts or decrypts AES using OFB block mode.

Encrypts or decrypts AES using OFB block mode. AES OFB 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 – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- input – Input data for OFB block mode

- output – **[out]** Output data for OFB block mode
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first input block.

Returns
Status from encrypt operation

status_t HASHCRYPT_AES_EncryptCfb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *plaintext, uint8_t *ciphertext, size_t size, const uint8_t iv[16])

Encrypts AES using CFB block mode.

Parameters

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

Returns
Status from encrypt operation

status_t HASHCRYPT_AES_DecryptCfb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle, const uint8_t *ciphertext, uint8_t *plaintext, size_t size, const uint8_t iv[16])

Decrypts AES using CFB block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plaintext 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

HASHCRYPT_AES_BLOCK_SIZE

AES block size in bytes

AES_ENCRYPT

AES_DECRYPT

struct _hashcrypt_handle

#include <fsl_hashcrypt.h> Specify HASHCRYPT's key resource.

Public Members

uint32_t keyWord[8]

Copy of user key (set by HASHCRYPT_AES_SetKey().

hashcrypt_key_t keyType
For operations with key (such as AES encryption/decryption), specify key type.

2.18 Hashcrypt HASH

typedef struct *_hashcrypt_hash_ctx_t* hashcrypt_hash_ctx_t
Storage type used to save hash context.

typedef void (*hashcrypt_callback_t)(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, *status_t* status, void *userData)

HASHCRYPT background hash callback function.

status_t HASHCRYPT_SHA(HASHCRYPT_Type *base, *hashcrypt_algo_t* algo, const uint8_t *input, size_t inputSize, uint8_t *output, size_t *outputSize)

Create HASH on given data.

Perform the full SHA in one function call. The function is blocking.

Parameters

- base – HASHCRYPT peripheral base address
- algo – Underlaying algorithm to use for hash computation.
- input – Input data
- inputSize – Size of input data in bytes
- output – **[out]** Output hash data
- outputSize – **[out]** Output parameter storing the size of the output hash in bytes

Returns
Status of the one call hash operation.

status_t HASHCRYPT_SHA_Init(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, *hashcrypt_algo_t* algo)

Initialize HASH context.

This function initializes the HASH.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[out]** Output hash context
- algo – Underlaying algorithm to use for hash computation.

Returns
Status of initialization

status_t HASHCRYPT_SHA_Update(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, const uint8_t *input, size_t inputSize)

Add data to current HASH.

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed. The functions blocks. If it returns kStatus_Success, the running hash has been updated (HASHCRYPT has processed the input data), so the memory at input pointer can be released back to system. The HASHCRYPT context buffer is updated with the running hash and with all necessary information to support possible context switch.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** HASH context
- input – Input data
- inputSize – Size of input data in bytes

Returns
Status of the hash update operation

status_t HASHCRYPT_SHA_Finish(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, uint8_t *output, size_t *outputSize)

Finalize hashing.

Outputs the final hash (computed by HASHCRYPT_HASH_Update()) and erases the context.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** Input hash context
- output – **[out]** Output hash data
- outputSize – **[inout]** Optional parameter (can be passed as NULL). On function entry, it specifies the size of output[] buffer. On function return, it stores the number of updated output bytes.

Returns
Status of the hash finish operation

HASHCRYPT_HASH_CTX_SIZE

HASHCRYPT HASH Context size.

struct _hashcrypt_hash_ctx_t

#include <fsl_hashcrypt.h> Storage type used to save hash context.

Public Members

uint32_t x[22]
storage

2.19 I2C: Inter-Integrated Circuit Driver

2.20 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.

2.21 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

2.22 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.

2.23 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.

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.

2.24 I2S: I2S Driver

2.25 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

2.26 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

2.27 INPUTMUX: Input Multiplexing Driver

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 INPUTMUX registers with same name.
- connection – Applies signal from source signals collection to target signal.

Return values

None. –

void INPUTMUX_EnableSignal(void *base, inputmux_signal_t signal, bool enable)

Enable/disable a signal.

This function gates the INPUTPMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.
- signal – Enable signal register id and bit offset.
- enable – Selects enable or disable.

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. –

2.28 IAP_KBP Driver

ROM API status codes.

Values:

enumerator kStatus_RomApiExecuteCompleted

ROM successfully process the whole sb file/boot image.

enumerator kStatus_RomApiNeedMoreData

ROM needs more data to continue processing the boot image.

enumerator kStatus_RomApiBufferSizeNotEnough

The user buffer is not enough for use by Kboot during execution of the operation.

enumerator kStatus_RomApiInvalidBuffer

The user buffer is not ok for sbloader or authentication.

enum _kb_operation

Details of the operation to be performed by the ROM.

The kRomAuthenticateImage operation requires the entire signed image to be available to the application.

Values:

enumerator kRomAuthenticateImage

Authenticate a signed image.

enumerator kRomLoadImage

Load SB file.

enumerator kRomOperationCount

enum _kb_security_profile

Security constraint flags, Security profile flags.

Values:

enumerator kKbootMinRSA4096

typedef enum *_kb_operation* kb_operation_t

Details of the operation to be performed by the ROM.

The kRomAuthenticateImage operation requires the entire signed image to be available to the application.

typedef struct *_kb_region* kb_region_t

Memory region definition.

typedef struct *_kb_load_sb* kb_load_sb_t

User-provided options passed into kb_init().

The buffer field is a pointer to memory provided by the caller for use by Kboot during execution of the operation. Minimum size is the size of each certificate in the chain plus 432 bytes additional per certificate.

The profile field is a mask that specifies which features are required in the SB file or image being processed. This includes the minimum AES and RSA key sizes. See the _kb_security_profile enum for profile mask constants. The image being loaded or authenticated must match the profile or an error will be returned.

minBuildNumber is an optional field that can be used to prevent version rollback. The API will check the build number of the image, and if it is less than minBuildNumber will fail with an error.

maxImageLength is used to verify the offsetToCertificateBlockHeaderInBytes value at the beginning of a signed image. It should be set to the length of the SB file. If verifying an image in flash, it can be set to the internal flash size or a large number like 0x10000000.

userRHK can optionally be used by the user to override the RHK in IFR. If userRHK is not NULL, it points to a 32-byte array containing the SHA-256 of the root certificate's RSA public key.

The regions field points to an array of memory regions that the SB file being loaded is allowed to access. If regions is NULL, then all memory is accessible by the SB file. This feature is required to prevent a malicious image from erasing good code or RAM contents while it

is being loaded, only for us to find that the image is inauthentic when we hit the end of the section.

overrideSBBootSectionID lets the caller override the default section of the SB file that is processed during a kKbootLoadSB operation. By default, the section specified in the firstBootableSectionID field of the SB header is loaded. If overrideSBBootSectionID is non-zero, then the section with the given ID will be loaded instead.

The userSBKEK field lets a user provide their own AES-256 key for unwrapping keys in an SB file during the kKbootLoadSB operation. userSBKEK should point to a 32-byte AES-256 key. If userSBKEK is NULL then the IFR SBKEK will be used. After kb_init() returns, the caller should zero out the data pointed to by userSBKEK, as the API will have installed the key in the CAU3.

typedef struct *_kb_authenticate* kb_authenticate_t

typedef struct *_kb_options* kb_options_t

typedef struct *_memory_region_interface* memory_region_interface_t

Interface to memory operations for one region of memory.

typedef struct *_memory_map_entry* memory_map_entry_t

Structure of a memory map entry.

typedef struct *_kb_opaque_session_ref* kb_session_ref_t

status_t kb_init(*kb_session_ref_t* **session, const *kb_options_t* *options)

Initialize ROM API for a given operation.

Inits the ROM API based on the options provided by the application in the second argument. Every call to rom_init() should be paired with a call to rom_deinit().

Return values

- kStatus_Success – API was executed successfully.
- kStatus_InvalidArgument – An invalid argument is provided.
- kStatus_RomApiBufferSizeNotEnough – The user buffer is not enough for use by Kboot during execution of the operation.
- kStatus_RomApiInvalidBuffer – The user buffer is not ok for sbloader or authentication.
- kStatus_SKBOOT_Fail – Return the failed status of secure boot.
- kStatus_SKBOOT_KeyStoreMarkerInvalid – The key code for the particular PRINCE region is not present in the keystore
- kStatus_SKBOOT_Success – Return the successful status of secure boot.

status_t kb_deinit(*kb_session_ref_t* *session)

Cleans up the ROM API context.

After this call, the context parameter can be reused for another operation by calling rom_init() again.

Return values

kStatus_Success – API was executed successfully

status_t kb_execute(*kb_session_ref_t* *session, const uint8_t *data, uint32_t dataLength)

Perform the operation configured during init.

This application must call this API repeatedly, passing in sequential chunks of data from the boot image (SB file) that is to be processed. The ROM will perform the selected operation on this data and return. The application may call this function with as much or as little data as it wishes, which can be used to select the granularity of time given to the application in between executing the operation.

Parameters

- session – Current ROM context pointer.
- data – Buffer of boot image data provided to the ROM by the application.
- dataLength – Length in bytes of the data in the buffer provided to the ROM.

Return values

- kStatus_Success – ROM successfully process the part of sb file/boot image.
- kStatus_RomApiExecuteCompleted – ROM successfully process the whole sb file/boot image.
- kStatus_Fail – An error occurred while executing the operation.
- kStatus_RomApiNeedMoreData – No error occurred, but the ROM needs more data to continue processing the boot image.
- kStatus_RomApiBufferSizeNotEnough – user buffer is not enough for use by Kboot during execution of the operation.

kStatusGroup_RomApi

ROM API status group number.

struct _kb_region

#include <fsl_iap_kbp.h> Memory region definition.

struct _kb_load_sb

#include <fsl_iap_kbp.h> User-provided options passed into kb_init().

The buffer field is a pointer to memory provided by the caller for use by Kboot during execution of the operation. Minimum size is the size of each certificate in the chain plus 432 bytes additional per certificate.

The profile field is a mask that specifies which features are required in the SB file or image being processed. This includes the minimum AES and RSA key sizes. See the _kb_security_profile enum for profile mask constants. The image being loaded or authenticated must match the profile or an error will be returned.

minBuildNumber is an optional field that can be used to prevent version rollback. The API will check the build number of the image, and if it is less than minBuildNumber will fail with an error.

maxImageLength is used to verify the offsetToCertificateBlockHeaderInBytes value at the beginning of a signed image. It should be set to the length of the SB file. If verifying an image in flash, it can be set to the internal flash size or a large number like 0x10000000.

userRHK can optionally be used by the user to override the RHK in IFR. If userRHK is not NULL, it points to a 32-byte array containing the SHA-256 of the root certificate's RSA public key.

The regions field points to an array of memory regions that the SB file being loaded is allowed to access. If regions is NULL, then all memory is accessible by the SB file. This feature is required to prevent a malicious image from erasing good code or RAM contents while it is being loaded, only for us to find that the image is inauthentic when we hit the end of the section.

overrideSBBootSectionID lets the caller override the default section of the SB file that is processed during a kKbootLoadSB operation. By default, the section specified in the firstBootableSectionID field of the SB header is loaded. If overrideSBBootSectionID is non-zero, then the section with the given ID will be loaded instead.

The userSBKEK field lets a user provide their own AES-256 key for unwrapping keys in an SB file during the kKbootLoadSB operation. userSBKEK should point to a 32-byte AES-256 key. If userSBKEK is NULL then the IFR SBKEK will be used. After kb_init() returns, the

caller should zero out the data pointed to by userSBKEK, as the API will have installed the key in the CAU3.

struct _kb_authenticate
#include <fsl_iap_kbp.h>

struct _kb_options
#include <fsl_iap_kbp.h>

Public Members

uint32_t version
Should be set to kKbootApiVersion.

uint8_t *buffer
Caller-provided buffer used by Kboot.

struct _memory_region_interface
#include <fsl_iap_kbp.h> Interface to memory operations for one region of memory.

struct _memory_map_entry
#include <fsl_iap_kbp.h> Structure of a memory map entry.

struct _kb_opaque_session_ref
#include <fsl_iap_kbp.h>

union ___unnamed11___

Public Members

kb_authenticate_t authenticate

kb_load_sb_t loadSB
Settings for kKbootAuthenticate operation.

2.29 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 specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)
: Clear the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)
: Toggle the bits specifiled 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 specifiled by *clearBits* and set the bits specifiled 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().

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.

2.30 LPADC: 12-bit SAR Analog-to-Digital Converter Driver

enum _lpadc_status_flags
Define hardware flags of the module.

Values:

enumerator kLPADC_ResultFIFO0OverflowFlag
Indicates that more data has been written to the Result FIFO 0 than it can hold.

enumerator kLPADC_ResultFIFO0ReadyFlag
Indicates when the number of valid datawords in the result FIFO 0 is greater than the setting watermark level.

enumerator kLPADC_TriggerExceptionFlag
Indicates that a trigger exception event has occurred.

enumerator kLPADC_TriggerCompletionFlag
Indicates that a trigger completion event has occurred.

enumerator kLPADC_CalibrationReadyFlag
Indicates that the calibration process is done.

enumerator kLPADC_ActiveFlag
Indicates that the ADC is in active state.

enumerator kLPADC_ResultFIFOOverflowFlag
To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowFlag as instead.

enumerator kLPADC_ResultFIFOReadyFlag
To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0ReadyFlag as instead.

enum _lpadc_interrupt_enable
Define interrupt switchers of the module.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_ResultFIFO0OverflowInterruptEnable
Configures ADC to generate overflow interrupt requests when FOF0 flag is asserted.

enumerator kLPADC_FIFO0WatermarkInterruptEnable
Configures ADC to generate watermark interrupt requests when RDY0 flag is asserted.

enumerator kLPADC_ResultFIFOOverflowInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowInterruptEnable as instead.

enumerator kLPADC_FIFOWatermarkInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_FIFO0WatermarkInterruptEnable as instead.

enumerator kLPADC_TriggerExceptionInterruptEnable

Configures ADC to generate trigger exception interrupt.

enumerator kLPADC_Trigger0CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 0 completion.

enumerator kLPADC_Trigger1CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 1 completion.

enumerator kLPADC_Trigger2CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 2 completion.

enumerator kLPADC_Trigger3CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 3 completion.

enumerator kLPADC_Trigger4CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 4 completion.

enumerator kLPADC_Trigger5CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 5 completion.

enumerator kLPADC_Trigger6CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 6 completion.

enumerator kLPADC_Trigger7CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 7 completion.

enumerator kLPADC_Trigger8CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 8 completion.

enumerator kLPADC_Trigger9CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 9 completion.

enumerator kLPADC_Trigger10CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 10 completion.

enumerator kLPADC_Trigger11CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 11 completion.

enumerator kLPADC_Trigger12CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 12 completion.

enumerator kLPADC_Trigger13CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 13 completion.

enumerator kLPADC_Trigger14CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 14 completion.

enumerator kLPADC_Trigger15CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 15 completion.

enum _lpadc_trigger_status_flags

The enumerator of lpadc trigger status flags, including interrupted flags and completed flags.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_Trigger0InterruptedFlag

Trigger 0 is interrupted by a high priority exception.

enumerator kLPADC_Trigger1InterruptedFlag

Trigger 1 is interrupted by a high priority exception.

enumerator kLPADC_Trigger2InterruptedFlag

Trigger 2 is interrupted by a high priority exception.

enumerator kLPADC_Trigger3InterruptedFlag

Trigger 3 is interrupted by a high priority exception.

enumerator kLPADC_Trigger4InterruptedFlag

Trigger 4 is interrupted by a high priority exception.

enumerator kLPADC_Trigger5InterruptedFlag

Trigger 5 is interrupted by a high priority exception.

enumerator kLPADC_Trigger6InterruptedFlag

Trigger 6 is interrupted by a high priority exception.

enumerator kLPADC_Trigger7InterruptedFlag

Trigger 7 is interrupted by a high priority exception.

enumerator kLPADC_Trigger8InterruptedFlag

Trigger 8 is interrupted by a high priority exception.

enumerator kLPADC_Trigger9InterruptedFlag

Trigger 9 is interrupted by a high priority exception.

enumerator kLPADC_Trigger10InterruptedFlag

Trigger 10 is interrupted by a high priority exception.

enumerator kLPADC_Trigger11InterruptedFlag

Trigger 11 is interrupted by a high priority exception.

enumerator kLPADC_Trigger12InterruptedFlag

Trigger 12 is interrupted by a high priority exception.

enumerator kLPADC_Trigger13InterruptedFlag

Trigger 13 is interrupted by a high priority exception.

enumerator kLPADC_Trigger14InterruptedFlag

Trigger 14 is interrupted by a high priority exception.

enumerator kLPADC_Trigger15InterruptedFlag

Trigger 15 is interrupted by a high priority exception.

enumerator kLPADC_Trigger0CompletedFlag

Trigger 0 is completed and trigger 0 has enabled completion interrupts.

enumerator kLPADC_Trigger1CompletedFlag

Trigger 1 is completed and trigger 1 has enabled completion interrupts.

enumerator kLPADC_Trigger2CompletedFlag
Trigger 2 is completed and trigger 2 has enabled completion interrupts.

enumerator kLPADC_Trigger3CompletedFlag
Trigger 3 is completed and trigger 3 has enabled completion interrupts.

enumerator kLPADC_Trigger4CompletedFlag
Trigger 4 is completed and trigger 4 has enabled completion interrupts.

enumerator kLPADC_Trigger5CompletedFlag
Trigger 5 is completed and trigger 5 has enabled completion interrupts.

enumerator kLPADC_Trigger6CompletedFlag
Trigger 6 is completed and trigger 6 has enabled completion interrupts.

enumerator kLPADC_Trigger7CompletedFlag
Trigger 7 is completed and trigger 7 has enabled completion interrupts.

enumerator kLPADC_Trigger8CompletedFlag
Trigger 8 is completed and trigger 8 has enabled completion interrupts.

enumerator kLPADC_Trigger9CompletedFlag
Trigger 9 is completed and trigger 9 has enabled completion interrupts.

enumerator kLPADC_Trigger10CompletedFlag
Trigger 10 is completed and trigger 10 has enabled completion interrupts.

enumerator kLPADC_Trigger11CompletedFlag
Trigger 11 is completed and trigger 11 has enabled completion interrupts.

enumerator kLPADC_Trigger12CompletedFlag
Trigger 12 is completed and trigger 12 has enabled completion interrupts.

enumerator kLPADC_Trigger13CompletedFlag
Trigger 13 is completed and trigger 13 has enabled completion interrupts.

enumerator kLPADC_Trigger14CompletedFlag
Trigger 14 is completed and trigger 14 has enabled completion interrupts.

enumerator kLPADC_Trigger15CompletedFlag
Trigger 15 is completed and trigger 15 has enabled completion interrupts.

enum _lpadc_sample_scale_mode
Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Values:

enumerator kLPADC_SamplePartScale
Use divided input voltage signal. (For scale select,please refer to the reference manual).

enumerator kLPADC_SampleFullScale
Full scale (Factor of 1).

enum _lpadc_sample_channel_mode

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Values:

enumerator kLPADC_SampleChannelSingleEndSideA

Single-end mode, only A-side channel is converted.

enumerator kLPADC_SampleChannelSingleEndSideB

Single-end mode, only B-side channel is converted.

enumerator kLPADC_SampleChannelDiffBothSideAB

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDiffBothSideBA

Differential mode, the ADC result is (CHnB-CHnA).

enumerator kLPADC_SampleChannelDiffBothSide

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDualSingleEndBothSide

Dual-Single-Ended Mode. Both A side and B side channels are converted independently.

enum _lpadc_hardware_average_mode

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

Values:

enumerator kLPADC_HardwareAverageCount1

Single conversion.

enumerator kLPADC_HardwareAverageCount2

2 conversions averaged.

enumerator kLPADC_HardwareAverageCount4

4 conversions averaged.

enumerator kLPADC_HardwareAverageCount8

8 conversions averaged.

enumerator kLPADC_HardwareAverageCount16

16 conversions averaged.

enumerator kLPADC_HardwareAverageCount32

32 conversions averaged.

enumerator kLPADC_HardwareAverageCount64

64 conversions averaged.

enumerator kLPADC_HardwareAverageCount128

128 conversions averaged.

enum _lpadc_sample_time_mode

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Values:

enumerator kLPADC_SampleTimeADCK3

3 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK5

5 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK7

7 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK11

11 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK19

19 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK35

35 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK67

69 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK131

131 ADCK cycles total sample time.

enum _lpadc_hardware_compare_mode

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Values:

enumerator kLPADC_HardwareCompareDisabled

Compare disabled.

enumerator kLPADC_HardwareCompareStoreOnTrue

Compare enabled. Store on true.

enumerator kLPADC_HardwareCompareRepeatUntilTrue

Compare enabled. Repeat channel acquisition until true.

enum _lpadc_conversion_resolution_mode

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

Values:

enumerator kLPADC_ConversionResolutionStandard

Standard resolution. Single-ended 12-bit conversion, Differential 13-bit conversion with 2's complement output.

enumerator kLPADC_ConversionResolutionHigh
High resolution. Single-ended 16-bit conversion; Differential 16-bit conversion with 2's complement output.

enum _lpadc_conversion_average_mode
Define enumeration of conversion averages mode.

Configure the converion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

Values:

enumerator kLPADC_ConversionAverage1
Single conversion.

enumerator kLPADC_ConversionAverage2
2 conversions averaged.

enumerator kLPADC_ConversionAverage4
4 conversions averaged.

enumerator kLPADC_ConversionAverage8
8 conversions averaged.

enumerator kLPADC_ConversionAverage16
16 conversions averaged.

enumerator kLPADC_ConversionAverage32
32 conversions averaged.

enumerator kLPADC_ConversionAverage64
64 conversions averaged.

enumerator kLPADC_ConversionAverage128
128 conversions averaged.

enumerator kLPADC_ConversionAverageMax

enum _lpadc_reference_voltage_mode
Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

Values:

enumerator kLPADC_ReferenceVoltageAlt1
Option 1 setting.

enumerator kLPADC_ReferenceVoltageAlt2
Option 2 setting.

enumerator kLPADC_ReferenceVoltageAlt3
Option 3 setting.

enum _lpadc_power_level_mode
Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Values:

enumerator kLPADC_PowerLevelAlt1
Lowest power setting.

enumerator kLPADC_PowerLevelAlt2
Next lowest power setting.

enumerator kLPADC_PowerLevelAlt3
...

enumerator kLPADC_PowerLevelAlt4
Highest power setting.

enum _lpadc_offset_calibration_mode
Define enumeration of offset calibration mode.

Values:

enumerator kLPADC_OffsetCalibration12bitMode
12 bit offset calibration mode.

enumerator kLPADC_OffsetCalibration16bitMode
16 bit offset calibration mode.

enum _lpadc_trigger_priority_policy
Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

Values:

enumerator kLPADC_ConvPreemptImmediatelyNotAutoResumed
If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion is not automatically resumed or restarted.

enumerator kLPADC_ConvPreemptSoftlyNotAutoResumed
If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion is not resumed or restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoRestarted
If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptSoftlyAutoRestarted
If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be resumed.

enumerator kLPADC_ConvPreemptSoftlyAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will be automatically be resumed.

enumerator kLPADC_TriggerPriorityPreemptImmediately

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityPreemptSoftly

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityExceptionDisabled

High priority trigger exception disabled.

enum _lpadc_tune_value

Define enumeration of tune value.

Values:

enumerator kLPADC_TuneValue0

Tune value 0.

enumerator kLPADC_TuneValue1

Tune value 1.

enumerator kLPADC_TuneValue2

Tune value 2.

enumerator kLPADC_TuneValue3

Tune value 3.

typedef enum *_lpadc_sample_scale_mode* lpadc_sample_scale_mode_t

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

typedef enum *_lpadc_sample_channel_mode* lpadc_sample_channel_mode_t

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

typedef enum *_lpadc_hardware_average_mode* lpadc_hardware_average_mode_t

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

typedef enum *_lpadc_sample_time_mode* lpadc_sample_time_mode_t

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

typedef enum *_lpadc_hardware_compare_mode* lpadc_hardware_compare_mode_t

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

typedef enum *_lpadc_conversion_resolution_mode* lpadc_conversion_resolution_mode_t

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

typedef enum *_lpadc_conversion_average_mode* lpadc_conversion_average_mode_t

Define enumeration of conversion averages mode.

Configure the converion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

typedef enum *_lpadc_reference_voltage_mode* lpadc_reference_voltage_source_t

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

typedef enum *_lpadc_power_level_mode* lpadc_power_level_mode_t

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

typedef enum *_lpadc_offset_calibration_mode* lpadc_offset_calibration_mode_t

Define enumeration of offset calibration mode.

typedef enum *_lpadc_trigger_priority_policy* lpadc_trigger_priority_policy_t

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

typedef enum *_lpadc_tune_value* lpadc_tune_value_t

Define enumeration of tune value.

typedef struct *_lpadc_calibration_value* lpadc_calibration_value_t

A structure of calibration value.

LPADC_CONVERSION_COMPLETE_TIMEOUT

Max loops to wait for LPADC conversion complete.

When doing calibration, driver will wait for the completion of conversion. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

LPADC_CALIBRATION_READY_TIMEOUT

Max loops to wait for LPADC calibration ready.

Before doing calibration, driver will wait for the calibration ready. This parameter defines how many loops to check the calibration ready. If defined as 0, driver will wait forever until ready.

LPADC_GAIN_CAL_READY_TIMEOUT

Max loops to wait for LPADC gain calibration GAIN_CAL ready.

Before doing calibration, driver will wait for the gain calibration GAIN_CAL ready. This parameter defines how many loops to check the gain calibration GAIN_CAL ready. If defined as 0, driver will wait forever until ready.

ADC_OFSTRIM_OFSTRIM_MAX

ADC_OFSTRIM_OFSTRIM_SIGN

LPADC_GET_ACTIVE_COMMAND_STATUS(statusVal)

Define the MACRO function to get command status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

LPADC_GET_ACTIVE_TRIGGER_STATUE(statusVal)

Define the MACRO function to get trigger status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

void LPADC_Init(ADC_Type *base, const *lpadc_config_t* *config)

Initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.
- config – Pointer to configuration structure. See “lpadc_config_t”.

void LPADC_GetDefaultConfig(*lpadc_config_t* *config)

Gets an available pre-defined settings for initial configuration.

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

```
config->enableInDozeMode       = true;
config->enableAnalogPreliminary = false;
config->powerUpDelay           = 0x80;
config->referenceVoltageSource  = kLPADC_ReferenceVoltageAlt1;
config->powerLevelMode          = kLPADC_PowerLevelAlt1;
config->triggerPriorityPolicy   = kLPADC_TriggerPriorityPreemptImmediately;
config->enableConvPause         = false;
config->convPauseDelay          = 0U;
config->FIFOWatermark           = 0U;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_Deinit(ADC_Type *base)

De-initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.

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

Switch on/off the LPADC module.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the module.

static inline void LPADC_DoResetFIFO(ADC_Type *base)

Do reset the conversion FIFO.

Parameters

- base – LPADC peripheral base address.

static inline void LPADC_DoResetConfig(ADC_Type *base)

Do reset the module's configuration.

Reset all ADC internal logic and registers, except the Control Register (ADCx_CTRL).

Parameters

- base – LPADC peripheral base address.

static inline uint32_t LPADC_GetStatusFlags(ADC_Type *base)

Get status flags.

Parameters

- base – LPADC peripheral base address.

Returns

status flags' mask. See to _lpadc_status_flags.

static inline void LPADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clear status flags.

Only the flags can be cleared by writing ADCx_STATUS register would be cleared by this API.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for flags to be cleared. See to _lpadc_status_flags.

static inline uint32_t LPADC_GetTriggerStatusFlags(ADC_Type *base)

Get trigger status flags to indicate which trigger sequences have been completed or interrupted by a high priority trigger exception.

Parameters

- base – LPADC peripheral base address.

Returns

The OR'ed value of _lpadc_trigger_status_flags.

static inline void LPADC_ClearTriggerStatusFlags(ADC_Type *base, uint32_t mask)

Clear trigger status flags.

Parameters

- base – LPADC peripheral base address.

- mask – The mask of trigger status flags to be cleared, should be the OR'ed value of _lpadc_trigger_status_flags.

static inline void LPADC_EnableInterrupts(ADC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to _lpadc_interrupt_enable.

static inline void LPADC_DisableInterrupts(ADC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to _lpadc_interrupt_enable.

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

Switch on/off the DMA trigger for FIFO watermark event.

Parameters

- base – LPADC peripheral base address.
- enable – Switcher to the event.

static inline uint32_t LPADC_GetConvResultCount(ADC_Type *base)

Get the count of result kept in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

Returns

The count of result kept in conversion FIFO.

bool LPADC_GetConvResult(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

Returns

Status whether FIFO entry is valid.

void LPADC_GetConvResultBlocking(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO using blocking method.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

void LPADC_SetConvTriggerConfig(ADC_Type *base, uint32_t triggerId, const *lpadc_conv_trigger_config_t* *config)

Configure the conversion trigger source.

Each programmable trigger can launch the conversion command in command buffer.

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- config – Pointer to configuration structure. See to lpadc_conv_trigger_config_t.

void LPADC_GetDefaultConvTriggerConfig(*lpadc_conv_trigger_config_t* *config)

Gets an available pre-defined settings for trigger's configuration.

This function initializes the trigger's configuration structure with an available settings. The default values are:

```
config->targetCommandId       = 0U;
config->delayPower            = 0U;
config->priority              = 0U;
config->channelAFIFOSelect    = 0U;
config->channelBFIFOSelect    = 0U;
config->enableHardwareTrigger = false;
```

Parameters

- config – Pointer to configuration structure.

static inline void LPADC_DoSoftwareTrigger(ADC_Type *base, uint32_t triggerIdMask)

Do software trigger to conversion command.

Parameters

- base – LPADC peripheral base address.
- triggerIdMask – Mask value for software trigger indexes, which count from zero.

static inline void LPADC_EnableHardwareTriggerCommandSelection(ADC_Type *base, uint32_t triggerId, bool enable)

Enable hardware trigger command selection.

This function will use the hardware trigger command from ADC_ETC.The trigger command is then defined by ADC hardware trigger command selection field in ADC_ETC->TRIGx_CHAINy_z_n[CSEL].

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- enable – True to enable or flase to disable.

void LPADC_SetConvCommandConfig(ADC_Type *base, uint32_t commandId, const *lpadc_conv_command_config_t* *config)

Configure conversion command.

Note: The number of compare value register on different chips is different, that is mean in some chips, some command buffers do not have the compare functionality.

Parameters

- base – LPADC peripheral base address.
- commandId – ID for command in command buffer. Typically, the available value range is 1 - 15.

- config – Pointer to configuration structure. See to lpadc_conv_command_config_t.

void LPADC_GetDefaultConvCommandConfig(*lpadc_conv_command_config_t* *config)

Gets an available pre-defined settings for conversion command's configuration.

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
config->sampleScaleMode             = kLPADC_SampleFullScale;
config->channelBScaleMode           = kLPADC_SampleFullScale;
config->sampleChannelMode           = kLPADC_SampleChannelSingleEndSideA;
config->channelNumber               = 0U;
config->channelBNumber              = 0U;
config->chainedNextCommandNumber    = 0U;
config->enableAutoChannelIncrement = false;
config->loopCount                   = 0U;
config->hardwareAverageMode         = kLPADC_HardwareAverageCount1;
config->sampleTimeMode              = kLPADC_SampleTimeADCK3;
config->hardwareCompareMode         = kLPADC_HardwareCompareDisabled;
config->hardwareCompareValueHigh    = 0U;
config->hardwareCompareValueLow     = 0U;
config->conversionResolutionMode    = kLPADC_ConversionResolutionStandard;
config->enableWaitTrigger           = false;
config->enableChannelB              = false;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_EnableCalibration(ADC_Type *base, bool enable)

Enable the calibration function.

When CALOFS is set, the ADC is configured to perform a calibration function anytime the ADC executes a conversion. Any channel selected is ignored and the value returned in the RESFIFO is a signed value between -31 and 31. -32 is not a valid and is never a returned value. Software should copy the lower 6- bits of the conversion result stored in the RESFIFO after a completed calibration conversion to the OFSTRIM field. The OFSTRIM field is used in normal operation for offset correction.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetValue(ADC_Type *base, uint32_t value)

Set proper offset value to trim ADC.

To minimize the offset during normal operation, software should read the conversion result from the RESFIFO calibration operation and write the lower 6 bits to the OFSTRIM register.

Parameters

- base – LPADC peripheral base address.
- value – Setting offset value.

status_t LPADC_DoAutoCalibration(ADC_Type *base)

Do auto calibration.

Calibration function should be executed before using converter in application. It used the software trigger and a dummy conversion, get the offset and write them into the OFSTRIM register. It called some of functional API including:

- LPADC_EnableCalibration(...)

- LPADC_SetOffsetValue(...)
- LPADC_SetConvCommandConfig(...)
- LPADC_SetConvTriggerConfig(...)

Parameters

- base – LPADC peripheral base address.
- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void LPADC_SetOffsetValue(ADC_Type *base, int16_t value)

Set trim value for offset.

Note: For 16-bit conversions, each increment is 1/2 LSB resulting in a programmable offset range of -256 LSB to 255.5 LSB; For 12-bit conversions, each increment is 1/32 LSB resulting in a programmable offset range of -16 LSB to 15.96875 LSB.

Parameters

- base – LPADC peripheral base address.
- value – Offset trim value, is a 10-bit signed value between -512 and 511.

static inline void LPADC_GetOffsetValue(ADC_Type *base, int16_t *pValue)

Get trim value of offset.

Parameters

- base – LPADC peripheral base address.
- pValue – Pointer to the variable in type of int16_t to store offset value.

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

Enable the offset calibration function.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetCalibrationMode(ADC_Type *base, *lpadc_offset_calibration_mode_t* mode)

Set offset calibration mode.

Parameters

- base – LPADC peripheral base address.
- mode – set offset calibration mode.see to lpadc_offset_calibration_mode_t .

status_t LPADC_DoOffsetCalibration(ADC_Type *base)

Do offset calibration.

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.

- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_PrepareAutoCalibration(ADC_Type *base)

Prepare auto calibration, LPADC_FinishAutoCalibration has to be called before using the LPADC. LPADC_DoAutoCalibration has been split in two API to avoid to be stuck too long in the function.

Parameters

- base – LPADC peripheral base address.

status_t LPADC_FinishAutoCalibration(ADC_Type *base)

Finish auto calibration start with LPADC_PrepareAutoCalibration.

Note: This feature is used for LPADC with CTRL[CALOFSMODE].

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_GetCalibrationValue(ADC_Type *base, *lpadc_calibration_value_t* *ptrCalibrationValue)

Get calibration value into the memory which is defined by invoker.

Note: Please note the ADC will be disabled temporary.

Note: This function should be used after finish calibration.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure, this memory block should be always powered on even in low power modes.

status_t LPADC_SetCalibrationValue(ADC_Type *base, const *lpadc_calibration_value_t* *ptrCalibrationValue)

Set calibration value into ADC calibration registers.

Note: Please note the ADC will be disabled temporary.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure which contains ADC's calibration value.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void LPADC_RequestHighSpeedModeTrim(ADC_Type *base)

Request high speed mode trim calculation.

Parameters

- base – LPADC peripheral base address.

static inline int8_t LPADC_GetHighSpeedTrimValue(ADC_Type *base)

Get high speed mode trim value, the result is a 5-bit signed value between -16 and 15.

Note: The high speed mode trim value is used to minimize offset for high speed conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The calculated high speed mode trim value.

static inline void LPADC_SetHighSpeedTrimValue(ADC_Type *base, int8_t trimValue)

Set high speed mode trim value.

Note: If is possible to set the trim value manually, but it is recommended to use the LPADC_RequestHighSpeedModeTrim.

Parameters

- base – LPADC peripheral base address.
- trimValue – The trim value to be set.

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

Enable/disable high speed conversion mode, if enabled conversions complete 2 or 3 ADCK cycles sooner compared to conversion cycle counts when high speed mode is disabled.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable high speed conversion mode:
  - **true** Enable high speed conversion mode;
  - **false** Disable high speed conversion mode.

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

Enable/disable an additional ADCK cycle to conversion.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable an additional ADCK cycle to conversion:
  - **true** Enable an additional ADCK cycle to conversion;
  - **false** Disable an additional ADCK cycle to conversion.

static inline void LPADC_SetTuneValue(ADC_Type *base, *lpadc_tune_value_t* tuneValue)

Set tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.

- tuneValue – The tune value to be set, please refer to lpadc_tune_value_t.

static inline *lpadc_tune_value_t* LPADC_GetTuneValue(ADC_Type *base)

Get tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The tune value, please refer to lpadc_tune_value_t.

FSL_LPADC_DRIVER_VERSION

LPADC driver version 2.9.5.

struct lpadc_config_t

#include <fsl_lpadc.h> LPADC global configuration.

This structure would used to keep the settings for initialization.

Public Members

bool enableInternalClock

Enables the internally generated clock source. The clock source is used in clock selection logic at the chip level and is optionally used for the ADC clock source.

bool enableVref1LowVoltage

If voltage reference option1 input is below 1.8V, it should be “true”. If voltage reference option1 input is above 1.8V, it should be “false”.

bool enableInDozeMode

Control system transition to Stop and Wait power modes while ADC is converting. When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

lpadc_conversion_average_mode_t conversionAverageMode

Auto-Calibration Averages.

bool enableAnalogPreliminary

ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).

uint32_t powerUpDelay

When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize. The startup delay count of (powerUpDelay * 4) ADCK cycles must result in a longer delay than the analog startup time.

lpadc_reference_voltage_source_t referenceVoltageSource

Selects the voltage reference high used for conversions.

lpadc_power_level_mode_t powerLevelMode

Power Configuration Selection.

lpadc_trigger_priority_policy_t triggerPriorityPolicy

Control how higher priority triggers are handled, see to lpadc_trigger_priority_policy_t.

bool enableConvPause

Enables the ADC pausing function. When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in “Compare Until True” configuration.

uint32_t convPauseDelay

Controls the duration of pausing during command execution sequencing. The pause delay is a count of (convPauseDelay*4) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

uint32_t FIFOWatermark

FIFOWatermark is a programmable threshold setting. When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

struct lpadc_conv_command_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion command.

Public Members

lpadc_sample_scale_mode_t sampleScaleMode

Sample scale mode.

lpadc_sample_scale_mode_t channelBScaleMode

Alternate channe B Scale mode.

lpadc_sample_channel_mode_t sampleChannelMode

Channel sample mode.

uint32_t channelNumber

Channel number, select the channel or channel pair.

uint32_t channelBNumber

Alternate Channel B number, select the channel.

uint32_t chainedNextCommandNumber

Selects the next command to be executed after this command completes. 1-15 is available, 0 is to terminate the chain after this command.

bool enableAutoChannelIncrement

Loop with increment: when disabled, the “loopCount” field selects the number of times the selected channel is converted consecutively; when enabled, the “loopCount” field defines how many consecutive channels are converted as part of the command execution.

uint32_t loopCount

Selects how many times this command executes before finish and transition to the next command or Idle state. Command executes LOOP+1 times. 0-15 is available.

lpadc_hardware_average_mode_t hardwareAverageMode

Hardware average selection.

lpadc_sample_time_mode_t sampleTimeMode

Sample time selection.

lpadc_hardware_compare_mode_t hardwareCompareMode

Hardware compare selection.

uint32_t hardwareCompareValueHigh

Compare Value High. The available value range is in 16-bit.

uint32_t hardwareCompareValueLow

Compare Value Low. The available value range is in 16-bit.

lpadc_conversion_resolution_mode_t conversionResolutionMode

Conversion resolution mode.

bool enableWaitTrigger

Wait for trigger assertion before execution: when disabled, this command will be automatically executed; when enabled, the active trigger must be asserted again before executing this command.

struct lpadc_conv_trigger_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion trigger.

Public Members

uint32_t targetCommandId

Select the command from command buffer to execute upon detect of the associated trigger event.

uint32_t delayPower

Select the trigger delay duration to wait at the start of servicing a trigger event. When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is 2^delayPower ADCK cycles. The available value range is 4-bit.

uint32_t priority

Sets the priority of the associated trigger source. If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

bool enableHardwareTrigger

Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not. THe software trigger is always available.

struct lpadc_conv_result_t

#include <fsl_lpadc.h> Define the structure to keep the conversion result.

Public Members

uint32_t commandIdSource

Indicate the command buffer being executed that generated this result.

uint32_t loopCountIndex

Indicate the loop count value during command execution that generated this result.

uint32_t triggerIdSource

Indicate the trigger source that initiated a conversion and generated this result.

uint16_t convValue

Data result.

struct _lpadc_calibration_value

#include <fsl_lpadc.h> A structure of calibration value.

2.31 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

2.32 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.

2.33 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

2.34 OSTIMER: OS Event Timer Driver

void OSTIMER_Init(OSTIMER_Type *base)
Initializes an OSTIMER by turning its bus clock on.

void OSTIMER_Deinit(OSTIMER_Type *base)
Deinitializes a OSTIMER instance.

This function shuts down OSTIMER bus clock

Parameters

- base – OSTIMER peripheral base address.

uint64_t OSTIMER_GrayToDecimal(uint64_t gray)

Translate the value from gray-code to decimal.

Parameters

- gray – The gray value input.

Returns

The decimal value.

static inline uint64_t OSTIMER_DecimalToGray(uint64_t dec)

Translate the value from decimal to gray-code.

Parameters

- dec – The decimal value.

Returns

The gray code of the input value.

uint32_t OSTIMER_GetStatusFlags(OSTIMER_Type *base)

Get OSTIMER status Flags.

This returns the status flag. Currently, only match interrupt flag can be got.

Parameters

- base – OSTIMER peripheral base address.

Returns

status register value

void OSTIMER_ClearStatusFlags(OSTIMER_Type *base, uint32_t mask)

Clear Status Interrupt Flags.

This clears intrrupt status flag. Currently, only match interrupt flag can be cleared.

Parameters

- base – OSTIMER peripheral base address.
- mask – Clear bit mask.

Returns

none

status_t OSTIMER_SetMatchRawValue(OSTIMER_Type *base, uint64_t count, *ostimer_callback_t* cb)

Set the match raw value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central EVTIMER. Please note that, the data format may be gray-code, if so, please using OSTIMER_SetMatchValue().

Parameters

- base – OSTIMER peripheral base address.
- count – OSTIMER timer match value.(Value may be gray-code format)
- cb – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

kStatus_Success – - Set match raw value and enable interrupt Successfully.

status_t OSTIMER_SetMatchValue(OSTIMER_Type *base, uint64_t count, *ostimer_callback_t* cb)

Set the match value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central OS TIMER.

Parameters

- base – OSTIMER peripheral base address.
- count – OSTIMER timer match value.(Value is decimal format, and this value will be translate to Gray code in API if the IP counter is gray encoded.)
- cb – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

kStatus_Success – - Set match raw value and enable interrupt Successfully.

static inline void OSTIMER_SetMatchRegister(OSTIMER_Type *base, uint64_t value)

Set value to OSTIMER MATCH register directly.

This function writes the input value to OSTIMER MATCH register directly, it does not touch any other registers. Note that, the data format is gray-code. The function OSTIMER_DecimalToGray could convert decimal value to gray code.

Parameters

- base – OSTIMER peripheral base address.
- value – OSTIMER timer match value (Value is gray-code format).

static inline uint64_t OSTIMER_GetMatchRegister(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get the match value from OSTIMER. The value of timer match is gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of match register, data format is gray code.

static inline uint64_t OSTIMER_GetMatchValue(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get a match value from OSTIMER.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of match register.

static inline void OSTIMER_EnableMatchInterrupt(OSTIMER_Type *base)

Enable the OSTIMER counter match interrupt.

Enable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- base – OSTIMER peripheral base address.

static inline void OSTIMER_DisableMatchInterrupt(OSTIMER_Type *base)

Disable the OSTIMER counter match interrupt.

Disable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- base – OSTIMER peripheral base address.

static inline uint64_t OSTIMER_GetCurrentTimerRawValue(OSTIMER_Type *base)

Get current timer raw count value from OSTIMER.

This function will get the timer count value from OS timer register. The raw value of timer count may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of OSTIMER, may be gray code format.

uint64_t OSTIMER_GetCurrentTimerValue(OSTIMER_Type *base)

Get current timer count value from OSTIMER.

This function will get a decimal timer count value. If the RAW value of timer count is gray code format, it will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of OSTIMER which will be formated to decimal value.

static inline uint64_t OSTIMER_GetCaptureRawValue(OSTIMER_Type *base)

Get the capture value from OSTIMER.

This function will get a captured value from OSTIMER. The Raw value of timer capture may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of capture register, data format may be gray code.

uint64_t OSTIMER_GetCaptureValue(OSTIMER_Type *base)

Get the capture value from OSTIMER.

This function will get a capture decimal-value from OSTIMER. If the RAW value of timer count is gray code format, it will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of capture register, data format is decimal.

void OSTIMER_HandleIRQ(OSTIMER_Type *base, *ostimer_callback_t* cb)

OS timer interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in OSTIMER_SetMatchValue()). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- base – OS timer peripheral base address.
- cb – callback scheduled for this instance of OS timer

Returns
none

FSL_OSTIMER_DRIVER_VERSION
OSTIMER driver version.

enum _ostimer_flags
OSTIMER status flags.

Values:

enumerator kOSTIMER_MatchInterruptFlag
Match interrupt flag bit, sets if the match value was reached.

typedef void (*ostimer_callback_t)(void)
ostimer callback function.

2.35 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

enumerator kPINT_SecPinInt0
Secure 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

enumerator kPINT_SecPatternMatchBSlice0
Bit slice 0

enum _pint_pmatch_bslice_cfg
PINT Pattern Match configuration type.

Values:

enumerator kPINT_PatternMatchAlways
Always Contributes to product term match

enumerator kPINT_PatternMatchStickyRise
Sticky Rising edge

enumerator kPINT_PatternMatchStickyFall
Sticky Falling edge

enumerator kPINT_PatternMatchStickyBothEdges
Sticky Rising or Falling edge

enumerator kPINT_PatternMatchHigh
High level

enumerator kPINT_PatternMatchLow
Low level

enumerator kPINT_PatternMatchNever
Never contributes to product term match

enumerator kPINT_PatternMatchBothEdges
Either rising or falling edge

typedef enum *_pint_pin_enable* pint_pin_enable_t
PINT Pin Interrupt enable type.

typedef enum *_pint_int* pint_pin_int_t
PINT Pin Interrupt type.

typedef enum *_pint_pmatch_input_src* pint_pmatch_input_src_t
PINT Pattern Match bit slice input source type.

typedef enum *_pint_pmatch_bslice* pint_pmatch_bslice_t
PINT Pattern Match bit slice type.

typedef enum *_pint_pmatch_bslice_cfg* pint_pmatch_bslice_cfg_t
PINT Pattern Match configuration type.

typedef struct *_pint_status* pint_status_t
PINT event status.

typedef void (*pint_cb_t)(*pint_pin_int_t* pintr, *pint_status_t* *status)
PINT Callback function.

typedef struct *_pint_pmatch_cfg* pint_pmatch_cfg_t

void PINT_Init(PINT_Type *base)
Initialize PINT peripheral.

This function initializes the PINT peripheral and enables the clock.

Parameters

- base – Base address of the PINT peripheral.

Return values
None. –

void PINT_SetCallback(PINT_Type *base, *pint_cb_t* callback)
Set PINT callback.

This function set the callback for PINT 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.

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.

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>

2.36 PLU: Programmable Logic Unit

void PLU_Init(PLU_Type *base)

Enable the PLU clock and reset the module.

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

Parameters

- base – PLU peripheral base address

void PLU_Deinit(PLU_Type *base)

Gate the PLU clock.

Parameters

- base – PLU peripheral base address

static inline void PLU_SetLutInputSource(PLU_Type *base, *plu_lut_index_t* lutIndex, *plu_lut_in_index_t* lutInIndex, *plu_lut_input_source_t* inputSrc)

Set Input source of LUT.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs. For each LUT, the slot associated with the output from LUTn itself is tied low.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see plu_lut_index_t typedef enumeration).
- lutInIndex – LUT input index (see plu_lut_in_index_t typedef enumeration).
- inputSrc – LUT input source (see plu_lut_input_source_t typedef enumeration).

static inline void PLU_SetOutputSource(PLU_Type *base, *plu_output_index_t* outputIndex, *plu_output_source_t* outputSrc)

Set Output source of PLU.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs.

Parameters

- base – PLU peripheral base address.
- outputIndex – PLU output index (see plu_output_index_t typedef enumeration).
- outputSrc – PLU output source (see plu_output_source_t typedef enumeration).

static inline void PLU_SetLutTruthTable(PLU_Type *base, *plu_lut_index_t* lutIndex, uint32_t truthTable)

Set Truth Table of LUT.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see plu_lut_index_t typedef enumeration).
- truthTable – Truth Table value.

static inline uint32_t PLU_ReadOutputState(PLU_Type *base)

Read the current state of the 8 designated PLU Outputs.

Note: The PLU bus clock must be re-enabled prior to reading the Outpus Register if PLU bus clock is shut-off.

Parameters

- base – PLU peripheral base address.

Returns

Current PLU output state value.

void PLU_GetDefaultWakeIntConfig(*plu_wakeint_config_t* *config)

Gets an available pre-defined settings for wakeup/interrupt control.

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

```
config->filterMode = kPLU_WAKEINT_FILTER_MODE_BYPASS;
config->clockSource = kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC;
```

Parameters

- config – Pointer to configuration structure.

void PLU_EnableWakeIntRequest(PLU_Type *base, uint32_t interruptMask, const *plu_wakeint_config_t* *config)

Enable PLU outputs wakeup/interrupt request.

This function enables Any of the eight selected PLU outputs to contribute to an asynchronous wake-up or an interrupt request.

Note: If a PLU_CLKIN is provided, the raw wake-up/interrupt request will be set on the rising-edge of the PLU_CLKIN whenever the raw request signal is high. This registered signal will be glitch-free and just use the default wakeint config by PLU_GetDefaultWakeIntConfig(). If not, have to specify the filter mode and clock source to eliminate the glitches caused by long and widely disparate delays through the network

of LUTs making up the PLU. This way may increase power consumption in low-power operating modes and inject delay before the wake-up/interrupt request is generated.

Parameters

- base – PLU peripheral base address.
- interruptMask – PLU interrupt mask (see _plu_interrupt_mask enumeration).
- config – Pointer to configuration structure (see plu_wakeint_config_t typedef enumeration)

static inline void PLU_LatchInterrupt(PLU_Type *base)

Latch an interrupt.

This function latches the interrupt and then it can be cleared with PLU_ClearLatchedInterrupt().

Note: This mode is not compatible with use of the glitch filter. If this bit is set, the FILTER MODE should be set to kPLU_WAKEINT_FILTER_MODE_BYPASS (Bypass Mode) and PLU_CLKIN should be provided. If this bit is set, the wake-up/interrupt request will be set on the rising-edge of PLU_CLKIN whenever the raw wake-up/interrupt signal is high. The request must be cleared by software.

Parameters

- base – PLU peripheral base address.

void PLU_ClearLatchedInterrupt(PLU_Type *base)

Clear the latched interrupt.

This function clears the wake-up/interrupt request flag latched by PLU_LatchInterrupt()

Note: It is not necessary for the PLU bus clock to be enabled in order to write-to or read-back this bit.

Parameters

- base – PLU peripheral base address.

FSL_PLU_DRIVER_VERSION

Version 2.2.1

enum _plu_lut_index

Index of LUT.

Values:

enumerator kPLU_LUT_0

5-input Look-up Table 0

enumerator kPLU_LUT_1

5-input Look-up Table 1

enumerator kPLU_LUT_2

5-input Look-up Table 2

enumerator kPLU_LUT_3

5-input Look-up Table 3

enumerator kPLU_LUT_4

5-input Look-up Table 4

enumerator kPLU_LUT_5

5-input Look-up Table 5

enumerator kPLU_LUT_6
5-input Look-up Table 6

enumerator kPLU_LUT_7
5-input Look-up Table 7

enumerator kPLU_LUT_8
5-input Look-up Table 8

enumerator kPLU_LUT_9
5-input Look-up Table 9

enumerator kPLU_LUT_10
5-input Look-up Table 10

enumerator kPLU_LUT_11
5-input Look-up Table 11

enumerator kPLU_LUT_12
5-input Look-up Table 12

enumerator kPLU_LUT_13
5-input Look-up Table 13

enumerator kPLU_LUT_14
5-input Look-up Table 14

enumerator kPLU_LUT_15
5-input Look-up Table 15

enumerator kPLU_LUT_16
5-input Look-up Table 16

enumerator kPLU_LUT_17
5-input Look-up Table 17

enumerator kPLU_LUT_18
5-input Look-up Table 18

enumerator kPLU_LUT_19
5-input Look-up Table 19

enumerator kPLU_LUT_20
5-input Look-up Table 20

enumerator kPLU_LUT_21
5-input Look-up Table 21

enumerator kPLU_LUT_22
5-input Look-up Table 22

enumerator kPLU_LUT_23
5-input Look-up Table 23

enumerator kPLU_LUT_24
5-input Look-up Table 24

enumerator kPLU_LUT_25
5-input Look-up Table 25

enum _plu_lut_in_index
Inputs of LUT. 5 input present for each LUT.

Values:

enumerator kPLU_LUT_IN_0
LUT input 0

enumerator kPLU_LUT_IN_1
LUT input 1

enumerator kPLU_LUT_IN_2
LUT input 2

enumerator kPLU_LUT_IN_3
LUT input 3

enumerator kPLU_LUT_IN_4
LUT input 4

enum _plu_lut_input_source
Available sources of LUT input.

Values:

enumerator kPLU_LUT_IN_SRC_PLU_IN_0
Select PLU input 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_1
Select PLU input 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_2
Select PLU input 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_3
Select PLU input 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_4
Select PLU input 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_5
Select PLU input 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_0
Select LUT output 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_1
Select LUT output 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_2
Select LUT output 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_3
Select LUT output 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_4
Select LUT output 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_5
Select LUT output 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_6
Select LUT output 6 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_7
Select LUT output 7 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_8
Select LUT output 8 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_9
Select LUT output 9 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_10
Select LUT output 10 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_11
Select LUT output 11 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_12
Select LUT output 12 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_13
Select LUT output 13 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_14
Select LUT output 14 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_15
Select LUT output 15 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_16
Select LUT output 16 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_17
Select LUT output 17 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_18
Select LUT output 18 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_19
Select LUT output 19 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_20
Select LUT output 20 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_21
Select LUT output 21 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_22
Select LUT output 22 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_23
Select LUT output 23 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_24
Select LUT output 24 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_25
Select LUT output 25 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_0
Select Flip-Flops state 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_1
Select Flip-Flops state 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_2
Select Flip-Flops state 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_3
Select Flip-Flops state 3 to be connected to LUTn Input x

enum _plu_output_index
PLU output multiplexer registers.

Values:

enumerator kPLU_OUTPUT_0
PLU OUTPUT 0

enumerator kPLU_OUTPUT_1
PLU OUTPUT 1

enumerator kPLU_OUTPUT_2
PLU OUTPUT 2

enumerator kPLU_OUTPUT_3
PLU OUTPUT 3

enumerator kPLU_OUTPUT_4
PLU OUTPUT 4

enumerator kPLU_OUTPUT_5
PLU OUTPUT 5

enumerator kPLU_OUTPUT_6
PLU OUTPUT 6

enumerator kPLU_OUTPUT_7
PLU OUTPUT 7

enum _plu_output_source
Available sources of PLU output.

Values:

enumerator kPLU_OUT_SRC_LUT_0
Select LUT0 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_1
Select LUT1 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_2
Select LUT2 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_3
Select LUT3 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_4
Select LUT4 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_5
Select LUT5 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_6
Select LUT6 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_7
Select LUT7 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_8
Select LUT8 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_9
Select LUT9 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_10
Select LUT10 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_11
Select LUT11 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_12
Select LUT12 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_13
Select LUT13 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_14
Select LUT14 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_15
Select LUT15 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_16
Select LUT16 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_17
Select LUT17 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_18
Select LUT18 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_19
Select LUT19 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_20
Select LUT20 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_21
Select LUT21 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_22
Select LUT22 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_23
Select LUT23 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_24
Select LUT24 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_25
Select LUT25 output to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_0
Select Flip-Flops state(0) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_1
Select Flip-Flops state(1) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_2
Select Flip-Flops state(2) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_3
Select Flip-Flops state(3) to be connected to PLU output

enum _plu_interrupt_mask

The enumerator of PLU Interrupt.

Values:

enumerator kPLU_OUTPUT_0_INTERRUPT_MASK

Select PLU output 0 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_1_INTERRUPT_MASK

Select PLU output 1 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_2_INTERRUPT_MASK

Select PLU output 2 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_3_INTERRUPT_MASK

Select PLU output 3 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_4_INTERRUPT_MASK

Select PLU output 4 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_5_INTERRUPT_MASK

Select PLU output 5 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_6_INTERRUPT_MASK

Select PLU output 6 contribute to interrupt/wake-up generation

enumerator kPLU_OUTPUT_7_INTERRUPT_MASK

Select PLU output 7 contribute to interrupt/wake-up generation

enum _plu_wakeint_filter_mode

Control input of the PLU, add filtering for glitch.

Values:

enumerator kPLU_WAKEINT_FILTER_MODE_BYPASS

Select Bypass mode

enumerator kPLU_WAKEINT_FILTER_MODE_1_CLK_PERIOD

Filter 1 clock period

enumerator kPLU_WAKEINT_FILTER_MODE_2_CLK_PERIOD

Filter 2 clock period

enumerator kPLU_WAKEINT_FILTER_MODE_3_CLK_PERIOD

Filter 3 clock period

enum _plu_wakeint_filter_clock_source

Clock source for filter mode.

Values:

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC

Select the 1MHz low-power oscillator as the filter clock

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_12MHZ_FRO

Select the 12MHz FRO as the filer clock

enumerator kPLU_WAKEINT_FILTER_CLK_SRC_ALT

Select a third clock source

typedef enum *_plu_lut_index* plu_lut_index_t

Index of LUT.

typedef enum *_plu_lut_in_index* plu_lut_in_index_t
Inputs of LUT. 5 input present for each LUT.

typedef enum *_plu_lut_input_source* plu_lut_input_source_t
Available sources of LUT input.

typedef enum *_plu_output_index* plu_output_index_t
PLU output multiplexer registers.

typedef enum *_plu_output_source* plu_output_source_t
Available sources of PLU output.

typedef enum *_plu_wakeint_filter_mode* plu_wakeint_filter_mode_t
Control input of the PLU, add filtering for glitch.

typedef enum *_plu_wakeint_filter_clock_source* plu_wakeint_filter_clock_source_t
Clock source for filter mode.

typedef struct *_plu_wakeint_config* plu_wakeint_config_t
Wake configuration.

struct __plu_wakeint_config
#include <fsl_plu.h> Wake configuration.

Public Members

plu_wakeint_filter_mode_t filterMode
Filter Mode.

plu_wakeint_filter_clock_source_t clockSource
The clock source for filter mode.

2.37 PRINCE: PRINCE bus crypto engine

FSL_PRINCE_DRIVER_VERSION
PRINCE driver version 2.6.0.

Current version: 2.6.0

Change log:

- Version 2.0.0
  - Initial version.
- Version 2.1.0
  - Update for the A1 rev. of LPC55Sxx serie.
- Version 2.2.0
  - Add runtime checking of the A0 and A1 rev. of LPC55Sxx serie to support both silicone revisions.
- Version 2.3.0
  - Add support for LPC55S1x and LPC55S2x series
- Version 2.3.0
  - Fix MISRA-2012 issues.
- Version 2.3.1

  - Add support for LPC55S0x series
- Version 2.3.2
  - Fix documentation of enumeration. Extend PRINCE example.
- Version 2.4.0
  - Add support for LPC55S3x series
- Version 2.5.0
  - Add PRINCE_Config() and PRINCE_Reconfig() features.
- Version 2.5.1
  - Fix build error due to renamed symbols
- Version 2.6.0
  - Renamed CSS to ELS

enum _skboot_status

Secure status enumeration.

Values:

enumerator kStatus_SKBOOT_Success

PRINCE Success

enumerator kStatus_SKBOOT_Fail

PRINCE Fail

enumerator kStatus_SKBOOT_InvalidArgument

PRINCE Invalid argument

enumerator kStatus_SKBOOT_KeyStoreMarkerInvalid

PRINCE Invalid marker

enum _secure_bool

Secure boolean enumeration.

Values:

enumerator kSECURE_TRUE

PRINCE true

enumerator kSECURE_FALSE

PRINCE false

enum _prince_region

Prince region.

Values:

enumerator kPRINCE_Region0

PRINCE region 0

enumerator kPRINCE_Region1

PRINCE region 1

enumerator kPRINCE_Region2

PRINCE region 2

enum _prince_lock

Prince lock.

Values:

enumerator kPRINCE_Region0Lock
PRINCE region 0 lock

enumerator kPRINCE_Region1Lock
PRINCE region 1 lock

enumerator kPRINCE_Region2Lock
PRINCE region 2 lock

enumerator kPRINCE_MaskLock
PRINCE mask register lock

enum _prince_flags
Prince flag.

Values:

enumerator kPRINCE_Flag_None
PRINCE Flag None

enumerator kPRINCE_Flag_EraseCheck
PRINCE Flag Erase check

enumerator kPRINCE_Flag_WriteCheck
PRINCE Flag Write check

typedef enum *_skboot_status* skboot_status_t
Secure status enumeration.

typedef enum *_secure_bool* secure_bool_t
Secure boolean enumeration.

typedef enum *_prince_region* prince_region_t
Prince region.

typedef enum *_prince_lock* prince_lock_t
Prince lock.

typedef enum *_prince_flags* prince_flags_t
Prince flag.

static inline void PRINCE_EncryptEnable(PRINCE_Type *base)
Enable data encryption.

This function enables PRINCE on-the-fly data encryption.

Parameters

- base – PRINCE peripheral address.

static inline void PRINCE_EncryptDisable(PRINCE_Type *base)
Disable data encryption.

This function disables PRINCE on-the-fly data encryption.

Parameters

- base – PRINCE peripheral address.

static inline bool PRINCE_IsEncryptEnable(PRINCE_Type *base)
Is Enable data encryption.

This function test if PRINCE on-the-fly data encryption is enabled.

Parameters

- base – PRINCE peripheral address.

Returns
true if enabled, false if not

static inline void PRINCE_SetMask(PRINCE_Type *base, uint64_t mask)

Sets PRINCE data mask.

This function sets the PRINCE mask that is used to mask decrypted data.

Parameters

- base – PRINCE peripheral address.
- mask – 64-bit data mask value.

static inline void PRINCE_SetLock(PRINCE_Type *base, uint32_t lock)

Locks access for specified region registers or data mask register.

This function sets lock on specified region registers or mask register.

Parameters

- base – PRINCE peripheral address.
- lock – registers to lock. This is a logical OR of members of the enumeration prince_lock_t

status_t PRINCE_GenNewIV(*prince_region_t* region, uint8_t *iv_code, bool store, *flash_config_t* *flash_context)

Generate new IV code.

This function generates new IV code and stores it into the persistent memory. Ensure about 800 bytes free space on the stack when calling this routine with the store parameter set to true!

Parameters

- region – PRINCE region index.
- iv_code – IV code pointer used for storing the newly generated 52 bytes long IV code.
- store – flag to allow storing the newly generated IV code into the persistent memory (FFR).
- flash_context – pointer to the flash driver context structure.

Returns
kStatus_Success upon success

Returns
kStatus_Fail otherwise, kStatus_Fail is also returned if the key code for the particular PRINCE region is not present in the keystore (though new IV code has been provided)

status_t PRINCE_LoadIV(*prince_region_t* region, uint8_t *iv_code)

Load IV code.

This function enables IV code loading into the PRINCE bus encryption engine.

Parameters

- region – PRINCE region index.
- iv_code – IV code pointer used for passing the IV code.

Returns
kStatus_Success upon success

Returns
kStatus_Fail otherwise

status_t PRINCE_SetEncryptForAddressRange(*prince_region_t* region, uint32_t start_address, uint32_t length, *flash_config_t* *flash_context, bool regenerate_iv)

Allow encryption/decryption for specified address range.

This function sets the encryption/decryption for specified address range. The SR mask value for the selected Prince region is calculated from provided start_address and length parameters. This calculated value is OR'ed with the actual SR mask value and stored into the PRINCE SR_ENABLE register and also into the persistent memory (FFR) to be used after the device reset. It is possible to define several nonadjacent encrypted areas within one Prince region when calling this function repeatedly. If the length parameter is set to 0, the SR mask value is set to 0 and thus the encryption/decryption for the whole selected Prince region is disabled. Ensure about 800 bytes free space on the stack when calling this routine!

Parameters

- region – PRINCE region index.
- start_address – start address of the area to be encrypted/decrypted.
- length – length of the area to be encrypted/decrypted.
- flash_context – pointer to the flash driver context structure.
- regenerate_iv – flag to allow IV code regenerating, storing into the persistent memory (FFR) and loading into the PRINCE engine

Returns
kStatus_Success upon success

Returns
kStatus_Fail otherwise

status_t PRINCE_GetRegionSREnable(PRINCE_Type *base, *prince_region_t* region, uint32_t *sr_enable)

Gets the PRINCE Sub-Region Enable register.

This function gets PRINCE SR_ENABLE register.

Parameters

- base – PRINCE peripheral address.
- region – PRINCE region index.
- sr_enable – Sub-Region Enable register pointer.

Returns
kStatus_Success upon success

Returns
kStatus_InvalidArgument

status_t PRINCE_GetRegionBaseAddress(PRINCE_Type *base, *prince_region_t* region, uint32_t *region_base_addr)

Gets the PRINCE region base address register.

This function gets PRINCE BASE_ADDR register.

Parameters

- base – PRINCE peripheral address.
- region – PRINCE region index.
- region_base_addr – Region base address pointer.

Returns
kStatus_Success upon success

Returns
kStatus_InvalidArgument

status_t PRINCE_SetRegionIV(PRINCE_Type *base, *prince_region_t* region, const uint8_t iv[8])

Sets the PRINCE region IV.

This function sets specified AES IV for the given region.

Parameters

- base – PRINCE peripheral address.
- region – Selection of the PRINCE region to be configured.
- iv – 64-bit AES IV in little-endian byte order.

status_t PRINCE_SetRegionBaseAddress(PRINCE_Type *base, *prince_region_t* region, uint32_t region_base_addr)

Sets the PRINCE region base address.

This function configures PRINCE region base address.

Parameters

- base – PRINCE peripheral address.
- region – Selection of the PRINCE region to be configured.
- region_base_addr – Base Address for region.

status_t PRINCE_SetRegionSREnable(PRINCE_Type *base, *prince_region_t* region, uint32_t sr_enable)

Sets the PRINCE Sub-Region Enable register.

This function configures PRINCE SR_ENABLE register.

Parameters

- base – PRINCE peripheral address.
- region – Selection of the PRINCE region to be configured.
- sr_enable – Sub-Region Enable register value.

status_t PRINCE_FlashEraseWithChecker(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)

Erases the flash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length. It deals with the flash erase function complenentary to the standard erase API of the IAP1 driver. This implementation additionally checks if the whole encrypted PRINCE subregions are erased at once to avoid secrets revealing. The checker implementation is limited to one contiguous PRINCE-controlled memory area.

Parameters

- config – The pointer to the flash driver context structure.
- start – The start address of the desired flash memory to be erased. The start address needs to be prince-sburegion-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words) to be erased. Must be prince-sburegion-size-aligned.
- key – The value used to validate all flash erase APIs.

Returns
kStatus_FLASH_Success API was executed successfully.

Returns
kStatus_FLASH_InvalidArgument An invalid argument is provided.

Returns
kStatus_FLASH_AlignmentError The parameter is not aligned with the specified baseline.

Returns
kStatus_FLASH_AddressError The address is out of range.

Returns
kStatus_FLASH_EraseKeyError The API erase key is invalid.

Returns
kStatus_FLASH_CommandFailure Run-time error during the command execution.

Returns
kStatus_FLASH_CommandNotSupported Flash API is not supported.

Returns
kStatus_FLASH_EccError A correctable or uncorrectable error during command execution.

Returns
kStatus_FLASH_EncryptedRegionsEraseNotDoneAtOnce Encrypted flash subregions are not erased at once.

status_t PRINCE_FlashProgramWithChecker(*flash_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length. It deals with the flash program function complenentary to the standard program API of the IAP1 driver. This implementation additionally checks if the whole PRINCE subregions are programmed at once to avoid secrets revealing. The checker implementation is limited to one contiguous PRINCE-controlled memory area.

Parameters

- config – The pointer to the flash driver context structure.
- start – The start address of the desired flash memory to be programmed. Must be prince-sburegion-aligned.
- src – A pointer to the source buffer of data that is to be programmed into the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be prince-sburegion-size-aligned.

Returns
kStatus_FLASH_Success API was executed successfully.

Returns
kStatus_FLASH_InvalidArgument An invalid argument is provided.

Returns
kStatus_FLASH_AlignmentError Parameter is not aligned with the specified baseline.

Returns
kStatus_FLASH_AddressError Address is out of range.

Returns
kStatus_FLASH_AccessError Invalid instruction codes and out-of bounds addresses.

Returns
kStatus_FLASH_CommandFailure Run-time error during the command execution.

Returns
kStatus_FLASH_CommandFailure Run-time error during the command execution.

Returns
kStatus_FLASH_CommandNotSupported Flash API is not supported.

Returns
kStatus_FLASH_EccError A correctable or uncorrectable error during command execution.

Returns
kStatus_FLASH_SizeError Encrypted flash subregions are not programmed at once.

FSL_PRINCE_DRIVER_SUBREGION_SIZE_IN_KB

FSL_PRINCE_DRIVER_MAX_FLASH_ADDR

ALIGN_DOWN(x, a)

2.38 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>

2.39 RNG: Random Number Generator

FSL_RNG_DRIVER_VERSION

RNG driver version. Version 2.0.3.

Current version: 2.0.3

Change log:

- Version 2.0.0
  - Initial version
- Version 2.0.1
  - Fix MISRA C-2012 issue.
- Version 2.0.2
  - Add RESET_PeripheralReset function inside RNG_Init and RNG_Deinit functions.

- Version 2.0.3
  - Modified RNG_Init and RNG_GetRandomData functions, added rng_accumulateEntropy and rng_readEntropy functions.
  - These changes are reflecting recommended usage of RNG according to device UM.

void RNG_Init(RNG_Type *base)

Initializes the RNG.

This function initializes the RNG. When called, the RNG module and ring oscillator is enabled.

Parameters

- base – RNG base address

Returns

If successful, returns the kStatus_RNG_Success. Otherwise, it returns an error.

void RNG_Deinit(RNG_Type *base)

Shuts down the RNG.

This function shuts down the RNG.

Parameters

- base – RNG base address.

status_t RNG_GetRandomData(RNG_Type *base, void *data, size_t dataSize)

Gets random data.

This function gets random data from the RNG.

Parameters

- base – RNG base address.
- data – Pointer address used to store random data.
- dataSize – Size of the buffer pointed by the data parameter.

Returns

random data

static inline uint32_t RNG_GetRandomWord(RNG_Type *base)

Returns random 32-bit number.

This function gets random number from the RNG.

Parameters

- base – RNG base address.

Returns

random number

2.40 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_EnableSubsecCounter(RTC_Type *base, bool enable)

Enable the RTC Sub-second counter (32KHZ).

Note: Only enable sub-second counter after RTC_ENA bit has been set to 1.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC sub-second counter.
  - true: Enable RTC sub-second counter.
  - false: Disable RTC sub-second counter.

static inline uint32_t RTC_GetSubsecValue(const RTC_Type *base)

A read of 32KHZ sub-seconds counter.

Parameters

- base – RTC peripheral base address

Returns

Current value of the SUBSEC register

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.

2.41 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 SCTIMER_SetupNextStateAction().

Parameters

- base – SCTimer peripheral base address

Returns

The current state

static inline void SCTIMER_SetCounterState(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t state)

Set the counter current state.

The function is to set the state variable bit field of STATE register. Writing to the STATE_L, STATE_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- state – The counter current state number (only support range from 0~31).

static inline uint16_t SCTIMER_GetCounterState(SCT_Type *base, *sctimer_counter_t* whichCounter)

Get the counter current state value.

The function is to get the state variable bit field of STATE register.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns
The the counter current state value.

status_t SCTIMER_SetupCaptureAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t *captureRegister, uint32_t event)

Setup capture of the counter value on trigger of a selected event.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- captureRegister – Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
- event – Event number that will trigger the capture

Returns
kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

void SCTIMER_SetCallback(SCT_Type *base, *sctimer_event_callback_t* callback, uint32_t event)

Receive 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 SCTIMER_SetupNextStateActionwithLdMethod

This transition will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition

static inline void SCTIMER_SetupEventActiveDirection(SCT_Type *base, *sctimer_event_active_direction_t* activeDirection, uint32_t event)

Setup event active direction when the counters are operating in BIDIR mode.

Parameters

- base – SCTimer peripheral base address
- activeDirection – Event generation active direction, see sctimer_event_active_direction_t.
- event – Event number that need setup the active direction.

static inline void SCTIMER_SetupOutputSetAction(SCT_Type *base, uint32_t whichIO, uint32_t event)

Set the Output.

This output will be set when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to set
- event – Event number that will trigger the output change

static inline void SCTIMER_SetupOutputClearAction(SCT_Type *base, uint32_t whichIO, uint32_t event)

Clear the Output.

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to clear
- event – Event number that will trigger the output change

void SCTIMER_SetupOutputToggleAction(SCT_Type *base, uint32_t whichIO, uint32_t event)

Toggle the output level.

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to toggle
- event – Event number that will trigger the output change

static inline void SCTIMER_SetupCounterLimitAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t event)

Limit the running counter.

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be limited

static inline void SCTIMER_SetupCounterStopAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t event)

Stop the running counter.

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be stopped

static inline void SCTIMER_SetupCounterStartAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t event)

Re-start the stopped counter.

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to re-start

static inline void SCTIMER_SetupCounterHaltAction(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t event)

Halt the running counter.

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the SCTIMER_StartTimer() function.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be halted

static inline void SCTIMER_SetupDmaTriggerAction(SCT_Type *base, uint32_t dmaNumber, uint32_t event)

Generate a DMA request.

DMA request will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- dmaNumber – The DMA request to generate
- event – Event number that will trigger the DMA request

static inline void SCTIMER_SetCOUNTValue(SCT_Type *base, *sctimer_counter_t* whichCounter, uint32_t value)

Set the value of counter.

The function is to set the value of Count register, Writing to the COUNT_L, COUNT_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- value – the counter value update to the COUNT register.

static inline uint32_t SCTIMER_GetCOUNTValue(SCT_Type *base, *sctimer_counter_t* whichCounter)

Get the value of counter.

The function is to read the value of Count register, software can read the counter registers at any time..

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The value of counter selected.

static inline void SCTIMER_SetEventInState(SCT_Type *base, uint32_t event, uint32_t state)

Set the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be set.
- state – The state value in which the event is enabled to occur.

static inline void SCTIMER_ClearEventInState(SCT_Type *base, uint32_t event, uint32_t state)

Clear the state mask bit field of EV_STATE register.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be clear.
- state – The state value in which the event is disabled to occur.

static inline bool SCTIMER_GetEventInState(SCT_Type *base, uint32_t event, uint32_t state)

Get the state mask bit field of EV_STATE register.

Note: This function is to check whether the event is enabled in a specific state.

Parameters

- base – SCTimer peripheral base address
- event – The EV_STATE register be read.
- state – The state value.

Returns

The the state mask bit field of EV_STATE register.

- true: The event is enable in state.
- false: The event is disable in state.

static inline uint32_t SCTIMER_GetCaptureValue(SCT_Type *base, *sctimer_counter_t* whichCounter, uint8_t capChannel)

Get the value of capture register.

This function returns the captured value upon occurrence of the events selected by the corresponding Capture Control registers occurred.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- capChannel – SCTimer capture register of capture channel.

Returns

The SCTimer counter value at which this register was last captured.

void SCTIMER_EventHandleIRQ(SCT_Type *base)

SCTimer interrupt handler.

Parameters

- base – SCTimer peripheral base address.

FSL_SCTIMER_DRIVER_VERSION

Version

enum _sctimer_pwm_mode
SCTimer PWM operation modes.

Values:

enumerator kSCTIMER_EdgeAlignedPwm
Edge-aligned PWM

enumerator kSCTIMER_CenterAlignedPwm
Center-aligned PWM

enum _sctimer_counter
SCTimer counters type.

Values:

enumerator kSCTIMER_Counter_L
16-bit Low counter.

enumerator kSCTIMER_Counter_H
16-bit High counter.

enumerator kSCTIMER_Counter_U
32-bit Unified counter.

enum _sctimer_input
List of SCTimer input pins.

Values:

enumerator kSCTIMER_Input_0
SCTIMER input 0

enumerator kSCTIMER_Input_1
SCTIMER input 1

enumerator kSCTIMER_Input_2
SCTIMER input 2

enumerator kSCTIMER_Input_3
SCTIMER input 3

enumerator kSCTIMER_Input_4
SCTIMER input 4

enumerator kSCTIMER_Input_5
SCTIMER input 5

enumerator kSCTIMER_Input_6
SCTIMER input 6

enumerator kSCTIMER_Input_7
SCTIMER input 7

enum _sctimer_out
List of SCTimer output pins.

Values:

enumerator kSCTIMER_Out_0
SCTIMER output 0

enumerator kSCTIMER_Out_1
SCTIMER output 1

enumerator kSCTIMER_Out_2
SCTIMER output 2

enumerator kSCTIMER_Out_3
SCTIMER output 3

enumerator kSCTIMER_Out_4
SCTIMER output 4

enumerator kSCTIMER_Out_5
SCTIMER output 5

enumerator kSCTIMER_Out_6
SCTIMER output 6

enumerator kSCTIMER_Out_7
SCTIMER output 7

enumerator kSCTIMER_Out_8
SCTIMER output 8

enumerator kSCTIMER_Out_9
SCTIMER output 9

enum _sctimer_pwm_level_select
SCTimer PWM output pulse mode: high-true, low-true or no output.

Values:

enumerator kSCTIMER_LowTrue
Low true pulses

enumerator kSCTIMER_HighTrue
High true pulses

enum _sctimer_clock_mode
SCTimer clock mode options.

Values:

enumerator kSCTIMER_System_ClockMode
System Clock Mode

enumerator kSCTIMER_Sampled_ClockMode
Sampled System Clock Mode

enumerator kSCTIMER_Input_ClockMode
SCT Input Clock Mode

enumerator kSCTIMER_Asynchronous_ClockMode
Asynchronous Mode

enum _sctimer_clock_select
SCTimer clock select options.

Values:

enumerator kSCTIMER_Clock_On_Rise_Input_0
Rising edges on input 0

enumerator kSCTIMER_Clock_On_Fall_Input_0
Falling edges on input 0

enumerator kSCTIMER_Clock_On_Rise_Input_1
Rising edges on input 1

enumerator kSCTIMER_Clock_On_Fall_Input_1
Falling edges on input 1

enumerator kSCTIMER_Clock_On_Rise_Input_2
Rising edges on input 2

enumerator kSCTIMER_Clock_On_Fall_Input_2
Falling edges on input 2

enumerator kSCTIMER_Clock_On_Rise_Input_3
Rising edges on input 3

enumerator kSCTIMER_Clock_On_Fall_Input_3
Falling edges on input 3

enumerator kSCTIMER_Clock_On_Rise_Input_4
Rising edges on input 4

enumerator kSCTIMER_Clock_On_Fall_Input_4
Falling edges on input 4

enumerator kSCTIMER_Clock_On_Rise_Input_5
Rising edges on input 5

enumerator kSCTIMER_Clock_On_Fall_Input_5
Falling edges on input 5

enumerator kSCTIMER_Clock_On_Rise_Input_6
Rising edges on input 6

enumerator kSCTIMER_Clock_On_Fall_Input_6
Falling edges on input 6

enumerator kSCTIMER_Clock_On_Rise_Input_7
Rising edges on input 7

enumerator kSCTIMER_Clock_On_Fall_Input_7
Falling edges on input 7

enum _sctimer_conflict_resolution
SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Values:

enumerator kSCTIMER_ResolveNone
No change

enumerator kSCTIMER_ResolveSet
Set output

enumerator kSCTIMER_ResolveClear
Clear output

enumerator kSCTIMER_ResolveToggle
Toggle output

enum _sctimer_event_active_direction

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

Values:

enumerator kSCTIMER_ActiveIndependent

This event is triggered regardless of the count direction.

enumerator kSCTIMER_ActiveInCountUp

This event is triggered only during up-counting when BIDIR = 1.

enumerator kSCTIMER_ActiveInCountDown

This event is triggered only during down-counting when BIDIR = 1.

enum _sctimer_event

List of SCTimer event types.

Values:

enumerator kSCTIMER_InputLowOrMatchEvent

enumerator kSCTIMER_InputRiseOrMatchEvent

enumerator kSCTIMER_InputFallOrMatchEvent

enumerator kSCTIMER_InputHighOrMatchEvent

enumerator kSCTIMER_MatchEventOnly

enumerator kSCTIMER_InputLowEvent

enumerator kSCTIMER_InputRiseEvent

enumerator kSCTIMER_InputFallEvent

enumerator kSCTIMER_InputHighEvent

enumerator kSCTIMER_InputLowAndMatchEvent

enumerator kSCTIMER_InputRiseAndMatchEvent

enumerator kSCTIMER_InputFallAndMatchEvent

enumerator kSCTIMER_InputHighAndMatchEvent

enumerator kSCTIMER_OutputLowOrMatchEvent

enumerator kSCTIMER_OutputRiseOrMatchEvent

enumerator kSCTIMER_OutputFallOrMatchEvent

enumerator kSCTIMER_OutputHighOrMatchEvent

enumerator kSCTIMER_OutputLowEvent

enumerator kSCTIMER_OutputRiseEvent

enumerator kSCTIMER_OutputFallEvent

enumerator kSCTIMER_OutputHighEvent

enumerator kSCTIMER_OutputLowAndMatchEvent

enumerator kSCTIMER_OutputRiseAndMatchEvent

enumerator kSCTIMER_OutputFallAndMatchEvent

enumerator kSCTIMER_OutputHighAndMatchEvent

enum _sctimer_interrupt_enable

List of SCTimer interrupts.

Values:

enumerator kSCTIMER_Event0InterruptEnable

Event 0 interrupt

enumerator kSCTIMER_Event1InterruptEnable

Event 1 interrupt

enumerator kSCTIMER_Event2InterruptEnable

Event 2 interrupt

enumerator kSCTIMER_Event3InterruptEnable

Event 3 interrupt

enumerator kSCTIMER_Event4InterruptEnable

Event 4 interrupt

enumerator kSCTIMER_Event5InterruptEnable

Event 5 interrupt

enumerator kSCTIMER_Event6InterruptEnable

Event 6 interrupt

enumerator kSCTIMER_Event7InterruptEnable

Event 7 interrupt

enumerator kSCTIMER_Event8InterruptEnable

Event 8 interrupt

enumerator kSCTIMER_Event9InterruptEnable

Event 9 interrupt

enumerator kSCTIMER_Event10InterruptEnable

Event 10 interrupt

enumerator kSCTIMER_Event11InterruptEnable

Event 11 interrupt

enumerator kSCTIMER_Event12InterruptEnable

Event 12 interrupt

enum _sctimer_status_flags

List of SCTimer flags.

Values:

enumerator kSCTIMER_Event0Flag

Event 0 Flag

enumerator kSCTIMER_Event1Flag

Event 1 Flag

enumerator kSCTIMER_Event2Flag

Event 2 Flag

enumerator kSCTIMER_Event3Flag

Event 3 Flag

enumerator kSCTIMER_Event4Flag
Event 4 Flag

enumerator kSCTIMER_Event5Flag
Event 5 Flag

enumerator kSCTIMER_Event6Flag
Event 6 Flag

enumerator kSCTIMER_Event7Flag
Event 7 Flag

enumerator kSCTIMER_Event8Flag
Event 8 Flag

enumerator kSCTIMER_Event9Flag
Event 9 Flag

enumerator kSCTIMER_Event10Flag
Event 10 Flag

enumerator kSCTIMER_Event11Flag
Event 11 Flag

enumerator kSCTIMER_Event12Flag
Event 12 Flag

enumerator kSCTIMER_BusErrorLFlag
Bus error due to write when L counter was not halted

enumerator kSCTIMER_BusErrorHFlag
Bus error due to write when H counter was not halted

typedef enum *_sctimer_pwm_mode* sctimer_pwm_mode_t
SCTimer PWM operation modes.

typedef enum *_sctimer_counter* sctimer_counter_t
SCTimer counters type.

typedef enum *_sctimer_input* sctimer_input_t
List of SCTimer input pins.

typedef enum *_sctimer_out* sctimer_out_t
List of SCTimer output pins.

typedef enum *_sctimer_pwm_level_select* sctimer_pwm_level_select_t
SCTimer PWM output pulse mode: high-true, low-true or no output.

typedef struct *_sctimer_pwm_signal_param* sctimer_pwm_signal_param_t
Options to configure a SCTimer PWM signal.

typedef enum *_sctimer_clock_mode* sctimer_clock_mode_t
SCTimer clock mode options.

typedef enum *_sctimer_clock_select* sctimer_clock_select_t
SCTimer clock select options.

typedef enum *_sctimer_conflict_resolution* sctimer_conflict_resolution_t
SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

typedef enum *_sctimer_event_active_direction* sctimer_event_active_direction_t

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

typedef enum *_sctimer_event* sctimer_event_t

List of SCTimer event types.

typedef void (*sctimer_event_callback_t)(void)

SCTimer callback typedef.

typedef enum *_sctimer_interrupt_enable* sctimer_interrupt_enable_t

List of SCTimer interrupts.

typedef enum *_sctimer_status_flags* sctimer_status_flags_t

List of SCTimer flags.

typedef struct *_sctimer_config* sctimer_config_t

SCTimer configuration structure.

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

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

SCT_EV_STATE_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);

2.42 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 card0 clock.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableCard1Clock(SDIF_Type *base, bool enable)

SDIF module enable/disable card1 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_EnableCard1LowPowerMode(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 card0 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.

static inline void SDIF_EnableCard1Power(SDIF_Type *base, bool enable)

enable/disable the card1 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 card0 data bus width

Parameters

- base – SDIF peripheral base address.
- type – data bus width type

void SDIF_SetCard1BusWidth(SDIF_Type *base, *sdif_bus_width_t* type)

set card1 data bus width

Parameters

- base – SDIF peripheral base address.
- type – data bus width type

static inline uint32_t SDIF_DetectCardInsert(SDIF_Type *base, bool data3)

SDIF module detect card0 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

static inline uint32_t SDIF_DetectCard1Insert(SDIF_Type *base, bool data3)

SDIF module detect card1 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.

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

2.43 skboot_authenticate

enum _skboot_status
SKBOOT return status.

Values:

enumerator kStatus_SKBOOT_Success
SKBOOT return success status.

enumerator kStatus_SKBOOT_Fail
SKBOOT return fail status.

enumerator kStatus_SKBOOT_InvalidArgument
SKBOOT return invalid argument status.

enumerator kStatus_SKBOOT_KeyStoreMarkerInvalid
SKBOOT return Keystore invalid Marker status.

enumerator kStatus_SKBOOT_HashcryptFinishedWithStatusSuccess
SKBOOT return Hashcrypt finished with the success status.

enumerator kStatus_SKBOOT_HashcryptFinishedWithStatusFail
SKBOOT return Hashcrypt finished with the fail status.

enum _secure_bool
Secure bool flag.

Values:

enumerator kSECURE_TRUE
Secure true flag.

enumerator kSECURE_FALSE
Secure false flag.

enumerator kSECURE_CALLPROTECT_SECURITY_FLAGS
Secure call protect the security flag.

enumerator kSECURE_CALLPROTECT_IS_APP_READY
Secure call protect the app is ready flag.

enumerator kSECURE_TRACKER_VERIFIED
Secure tracker verified flag.

typedef enum *_skboot_status* skboot_status_t
SKBOOT return status.

typedef enum *_secure_bool* secure_bool_t
Secure bool flag.

skboot_status_t skboot_authenticate(const uint8_t *imageStartAddr, *secure_bool_t* *isSignVerified)
Authenticate entry function with ARENA allocator init.

This is called by ROM boot or by ROM API g_skbootAuthenticateInterface

void HASH_IRQHandler(void)
Interface for image authentication API.

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 SYSCTL: I2S bridging and signal sharing Configuration

void SYSCTL_Init(SYSCTL_Type *base)
SYSCTL initial.

Parameters

- base – Base address of the SYSCTL peripheral.

void SYSCTL_Deinit(SYSCTL_Type *base)
SYSCTL deinit.

Parameters

- base – Base address of the SYSCTL peripheral.

void SYSCTL_SetFlexcommShareSet(SYSCTL_Type *base, uint32_t flexCommIndex, uint32_t sckSet, uint32_t wsSet, uint32_t dataInSet, uint32_t dataOutSet)
SYSCTL share set configure for flexcomm.

Parameters

- base – Base address of the SYSCTL peripheral.
- flexCommIndex – index of flexcomm, reference _sysctl_share_src
- sckSet – share set for sck,reference _sysctl_share_set_index
- wsSet – share set for ws, reference _sysctl_share_set_index
- dataInSet – share set for data in, reference _sysctl_share_set_index
- dataOutSet – share set for data out, reference _sysctl_dataout_mask

void SYSCTL_SetShareSet(SYSCTL_Type *base, uint32_t flexCommIndex, *sysctl_fcctrlsel_signal_t* signal, uint32_t set)
SYSCTL share set configure for separate signal.

Parameters

- base – Base address of the SYSCTL peripheral
- flexCommIndex – index of flexcomm,reference _sysctl_share_src
- signal – FCCTRLSEL signal shift
- set – share set for sck, reference _sysctl_share_set_index

void SYSCTL_SetShareSetSrc(SYSCTL_Type *base, uint32_t setIndex, uint32_t sckShareSrc, uint32_t wsShareSrc, uint32_t dataInShareSrc, uint32_t dataOutShareSrc)

SYSCTL share set source configure.

Parameters

- base – Base address of the SYSCTL peripheral
- setIndex – index of share set, reference _sysctl_share_set_index
- sckShareSrc – sck source for this share set,reference _sysctl_share_src
- wsShareSrc – ws source for this share set,reference _sysctl_share_src
- dataInShareSrc – data in source for this share set,reference _sysctl_share_src
- dataOutShareSrc – data out source for this share set,reference _sysctl_dataout_mask

void SYSCTL_SetShareSignalSrc(SYSCTL_Type *base, uint32_t setIndex, *sysctl_sharedctrlset_signal_t* signal, uint32_t shareSrc)

SYSCTL sck source configure.

Parameters

- base – Base address of the SYSCTL peripheral
- setIndex – index of share set, reference _sysctl_share_set_index
- signal – FCCTRLSEL signal shift
- shareSrc – sck source fro this share set,reference _sysctl_share_src

FSL_SYSCTL_DRIVER_VERSION

Group sysctl driver version for SDK.

Version 2.0.5.

enum _sysctl_share_set_index

SYSCTL share set.

Values:

enumerator kSYSCTL_ShareSet0

share set 0

enumerator kSYSCTL_ShareSet1

share set 1

enum _sysctl_fcctrlsel_signal

SYSCTL flexcomm signal.

Values:

enumerator kSYSCTL_FlexcommSignalSCK

SCK signal

enumerator kSYSCTL_FlexcommSignalWS

WS signal

enumerator kSYSCTL_FlexcommSignalDataIn
Data in signal

enumerator kSYSCTL_FlexcommSignalDataOut
Data out signal

enum _sysctl_share_src
SYSCTL flexcomm index.

Values:

enumerator kSYSCTL_Flexcomm0
share set 0

enumerator kSYSCTL_Flexcomm1
share set 1

enumerator kSYSCTL_Flexcomm2
share set 2

enumerator kSYSCTL_Flexcomm4
share set 4

enumerator kSYSCTL_Flexcomm5
share set 5

enumerator kSYSCTL_Flexcomm6
share set 6

enumerator kSYSCTL_Flexcomm7
share set 7

enum _sysctl_dataout_mask
SYSCTL shared data out mask.

Values:

enumerator kSYSCTL_Flexcomm0DataOut
share set 0

enumerator kSYSCTL_Flexcomm1DataOut
share set 1

enumerator kSYSCTL_Flexcomm2DataOut
share set 2

enumerator kSYSCTL_Flexcomm4DataOut
share set 4

enumerator kSYSCTL_Flexcomm5DataOut
share set 5

enumerator kSYSCTL_Flexcomm6DataOut
share set 6

enumerator kSYSCTL_Flexcomm7DataOut
share set 7

enum _sysctl_sharedctrlset_signal
SYSCTL flexcomm signal.

Values:

enumerator kSYSCTL_SharedCtrlSignalSCK
SCK signal

enumerator kSYSCTL_SharedCtrlSignalWS
WS signal

enumerator kSYSCTL_SharedCtrlSignalDataIn
Data in signal

enumerator kSYSCTL_SharedCtrlSignalDataOut
Data out signal

typedef enum *_sysctl_fcctrlsel_signal* sysctl_fcctrlsel_signal_t
SYSCTL flexcomm signal.

typedef enum *_sysctl_sharedctrlset_signal* sysctl_sharedctrlset_signal_t
SYSCTL flexcomm signal.

2.48 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.49 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

2.50 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 ___unnamed32___

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.

2.51 UTICK: MictoTick Timer Driver

void UTICK_Init(UTICK_Type *base)
Initializes an UTICK by turning its bus clock on.

void UTICK_Deinit(UTICK_Type *base)
Deinitializes a UTICK instance.

This function shuts down Utick bus clock

Parameters

- base – UTICK peripheral base address.

uint32_t UTICK_GetStatusFlags(UTICK_Type *base)
Get Status Flags.

This returns the status flag

Parameters

- base – UTICK peripheral base address.

Returns
status register value

void UTICK_ClearStatusFlags(UTICK_Type *base)

Clear Status Interrupt Flags.

This clears intr status flag

Parameters

- base – UTICK peripheral base address.

Returns

none

void UTICK_SetTick(UTICK_Type *base, *utick_mode_t* mode, uint32_t count, *utick_callback_t* cb)

Starts UTICK.

This function starts a repeat/onetime countdown with an optional callback

Parameters

- base – UTICK peripheral base address.
- mode – UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
- count – UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
- cb – UTICK callback (can be left as NULL if none, otherwise should be a void func(void))

Returns

none

void UTICK_HandleIRQ(UTICK_Type *base, *utick_callback_t* cb)

UTICK Interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in UTICK_SetTick()). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- base – UTICK peripheral base address.
- cb – callback scheduled for this instance of UTICK

Returns

none

FSL_UTICK_DRIVER_VERSION

UTICK driver version 2.0.5.

enum _utick_mode

UTICK timer operational mode.

Values:

enumerator kUTICK_Onetime

Trigger once

enumerator kUTICK_Repeat

Trigger repeatedly

typedef enum *_utick_mode* utick_mode_t

UTICK timer operational mode.

typedef void (*utick_callback_t)(void)

UTICK callback function.

2.52 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

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 File System

3.1.1 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.

3.2 Motor Control

3.2.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called "packet-driven BDM" interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT]((https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to FreeMASTER community or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the src/platforms directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called FMSTR_TRANSPORT with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The *mcuxsdk* folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a "middleware" component which may be downloaded along with the example applications from https://mcuxpresso.nxp.com/en/welcome.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- The official FreeMASTER middleware repository.
- Online version of this document

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer's physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample 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 FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as "automatic" in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- ***src/platforms*** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- ***src/common*** folder—contains the common driver source files shared by the driver for all supported platforms. All the *.c* files must be added to the project, compiled, and linked together with the application.
  - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
  - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
  - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
  - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
  - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
  - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
  - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
  - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
  - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

  - *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
  - *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
  - *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
  - *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
  - *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
  - *freemaster_serial.h* - defines the low-level character-oriented Serial API.
  - *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
  - *freemaster_can.h* - defines the low-level message-oriented CAN API.
  - *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
  - *freemaster_net.h* - definitions related to the Network transport.
  - *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
  - *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
  - *freemaster_utils.h* - definitions related to utility code.
- ***src/drivers/[sdk]/serial*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- ***src/drivers/[sdk]/can*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- ***src/drivers/[sdk]/network*** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
  - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
  - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR   [0|1]
#define FMSTR_SHORT_INTR  [0|1]
#define FMSTR_POLL_DRIVEN [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See *Driver interrupt modes*.

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- **FMSTR_SERIAL** - serial communication protocol
- **FMSTR_CAN** - using CAN communication
- **FMSTR_PDBDM** - using packet-driven BDM communication
- **FMSTR_NET** - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_SERIAL_DRV. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32...255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0...255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode.
The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the "True" single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE

```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call FMSTR_SetCanBaseAddress() to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options

FMSTR_DISABLE

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+©W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options

FMSTR_USE_SCOPE

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options

FMSTR_USE_RECORDER

```
#define FMSTR_USE_RECORDER [number]
```

Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.

Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature.
Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using FMSTR_RegisterAppCmdCall(). Default value is 0.

TSA options

FMSTR_USE_TSA

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool.
Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables.
Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project.
Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions.
Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used.
Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support.
The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr, FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_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_UINT*n*	Unsigned integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_SINT*n*	Signed integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_FRAC*n*	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 */
```

(continues on next page)

(continued from previous page)

```
/* The readme.txt file will be accessible at the fmstr://text_files/readme.txt URL */
FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt)) /* memory-mapped file */

/* Files can also be specified with a full path so the DIRECTORY entry does not apply */
FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index))          /* memory-mapped file */
FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) /* memory-mapped file */

/* Hyperlinks can point to a local MEMFILE object or to the Internet */
FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

/* Project file links simplify opening the projects from any URLs */
FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
FMSTR_TSA_PROJECT("Full Project (online)", "http://mycompany.com/prj/demo.pmp")

FMSTR_TSA_TABLE_END()
```

TSA API

FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR buffAddr, FMSTR_SIZE buffSize);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *buffAddr* [in] - address of the memory buffer for the dynamic TSA table
- *buffSize* [in] - size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

FMSTR_TsaAddVar

Prototype

```
FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR tsaName, FMSTR_TSATBL_STRPTR␣
↪tsaType,
    FMSTR_TSATBL_VOIDPTR varAddr, FMSTR_SIZE32 varSize,
    FMSTR_SIZE flags);
```

- Declaration: *freemaster.h*

- Implementation: *freemaster_tsa.c*

Arguments

- *tsaName* [in] - name of the object
- *tsaType* [in] - name of the object type
- *varAddr* [in] - address of the object
- *varSize* [in] - size of the object
- *flags* [in] - access flags; a combination of these values:
  - *FMSTR_TSA_INFO_RO_VAR* — read-only memory-mapped object (typically a variable)
  - *FMSTR_TSA_INFO_RW_VAR* — read/write memory-mapped object
  - *FMSTR_TSA_INFO_NON_VAR* — other entry, describing structure types, structure members, enumerations, and other types

Description This function can be called only when the dynamic TSA table is enabled by the FMSTR_USE_TSA_DYNAMIC configuration option and when the FMSTR_SetUpTsaBuff function call is made to assign the dynamic TSA table memory. This function adds an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See *TSA table definition* for more details about the TSA table entries.

Application Commands API

FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Description This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the FMSTR_APPCMDRESULT_NOCMD constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the FMSTR_AppCmdAck call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The FMSTR_GetAppCmd function does not report the commands for which a callback handler function exists. If the FMSTR_GetAppCmd function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns FMSTR_APPCMDRESULT_NOCMD.

FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* dataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *dataLen* [out] - pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see *FMSTR_GetAppCmd*).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the FMSTR_AppCmdAck call, copy the data out to a private buffer.

FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT resultCode);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultCode* [in] - the result code which is to be returned to FreeMASTER

Description This function is used when the Application Command processing finishes in the application. The resultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(FMSTR_ADDR resultDataAddr, FMSTR_SIZE resultDataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultDataAddr* [in] - pointer to the data buffer that is to be copied to the Application Command data buffer
- *resultDataLen* [in] - length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description This function can be used before the Application Command processing finishes, when there are data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use FMSTR_GetAppCmdData and the data buffer after FMSTR_AppCmdSetResponseData is called.

Note: The current version of FreeMASTER does not support the Application Command response data.

FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(FMSTR_APPCMD_CODE appCmdCode, FMSTR_
↪PAPPCMDFUNC callbackFunc);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *appCmdCode* [in] - the Application Command code for which the callback is to be registered
- *callbackFunc* [in] - pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return value This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when trying to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is

```
FMSTR_APPCMD_RESULT HandlerFunction(FMSTR_APPCMD_CODE nAppcmd,
    FMSTR_APPCMD_PDATA pData, FMSTR_SIZE nDataLen);
```

Where:

- *nAppcmd* -Application Command code
- *pData* —points to the Application Command data received (if any)
- *nDataLen* —information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

Note: The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the FMSTR_RegisterAppCmdCall function always fails.

Pipes API

FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT pipePort, FMSTR_PPIPEFUNC pipeCallback,
↪
    FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize,
    FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize,
    FMSTR_U8 type, const FMSTR_CHAR *name);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipePort* [in] - port number that identifies the pipe for the client
- *pipeCallback* [in] - pointer to the callback function that is called whenever a pipe data status changes
- *pipeRxBuff* [in] - address of the receive memory buffer
- *pipeRxSize* [in] - size of the receive memory buffer
- *pipeTxBuff* [in] - address of the transmit memory buffer
- *pipeTxSize* [in] - size of the transmit memory buffer
- *type* [in] - a combination of FMSTR_PIPE_MODE_xxx and FMSTR_PIPE_SIZE_xxx constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.
- *name* [in] - user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

Description This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the FMSTR_PipeRead call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the FMSTR_PipeWrite call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The *PipeHandler* name is only a placeholder and must be defined by the application.

```
void PipeHandler(FMSTR_HPIPE pipeHandle);
```

FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call

Description This function de-initializes the pipe object. No data can be received or sent on the pipe after this call.

FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data to be written
- *pipeDataLen* [in] - length of the data to be written
- *writeGranularity* [in] - size of the minimum unit of data which is to be written

Description This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the nGranularity value equal to the nLength value, all data are considered as one chunk which is either written successfully as a whole or not at all. The nGranularity value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE readGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data buffer to be filled with the received data
- *pipeDataLen* [in] - length of the data to be read
- *readGranularity* [in] - size of the minimum unit of data which is to be read

Description This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The readGranularity argument can be used to copy the data in larger chunks in the same way as described in the FMSTR_PipeWrite function.

API data types This section describes the data types used in the FreeMASTER driver. The information provided here can be useful when modifying or porting the FreeMASTER Communication Driver to new NXP platforms.

Note: The licensing conditions prohibit use of FreeMASTER and the FreeMASTER Communication Driver with non-NXP MPU or MCU products.

Public common types The table below describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Type name	Description
FMSTR_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.	
FMSTR_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.	
FMSTR_BOOL	Data type used as a general boolean type.
This type is used only in zero/non-zero conditions in the driver code.	
FMSTR_APPCM	Data type used to hold the Application Command code.
Generally, this is an unsigned 8-bit value.	
FMSTR_APPCM	Data type used to create the Application Command data buffer.
Generally, this is an unsigned 8-bit value.	
FMSTR_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.

FMSTR_TSA_TII	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 FMSTR_SIZE.	
FMSTR_TSA_TS	Data type used to hold a memory block size, as used in the TSA descriptors.
By default, this is defined as FMSTR_SIZE.	

Public Pipes types The table describes the data types used by the FreeMASTER Pipes API:

FMSTR_HPIPE	Pipe handle that identifies the open-pipe object.
Generally, this is a pointer to a void type.	
FMSTR_PIPE_P(	Integer type required to hold at least 7 bits of data.
Generally, this is an unsigned 8-bit or 16-bit type.	
FMSTR_PIPE_SI	Integer type required to hold at least 16 bits of data.
This is used to store the data buffer sizes.	
FMSTR_PPIPEF	Pointer to the pipe handler function.
See *FMSTR_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