Document Number: MCUXSDKAPIRM
Rev 2.11.0
Jan 2022

# MCUXpresso SDK API Reference Manual

NXP Semiconductors

# Contents

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
|---|---|---|

| Section No. | Title | Page No. |
| --- | --- | --- |

# Chapter 1
# Introduction

The MCUXpresso Software Development Kit (MCUXpresso SDK) is a collection of software enablement for NXP Microcontrollers that includes peripheral drivers, multicore support and integrated RTOS support for FreeRTOS$^{TM}$ . In addition to the base enablement, the MCUXpresso SDK is augmented with demo applications, driver example projects, and API documentation to help users quickly leverage the support provided by MCUXpresso SDK. The `MCUXpresso SDK Web Builder` is available to provide access to all MCUXpresso SDK packages. See the *MCUXpresso Software Development Kit (SD-K) Release Notes* (document MCUXSDKRN) in the Supported Devices section at `MCUXpresso-SDK: Software Development Kit for MCUXpresso` for details.

The MCUXpresso SDK is built with the following runtime software components:

- Arm® and DSP standard libraries, and CMSIS-compliant device header files which provide direct access to the peripheral registers.
- Peripheral drivers that provide stateless, high-performance, ease-of-use APIs. Communication drivers provide higher-level transactional APIs for a higher-performance option.
- RTOS wrapper driver built on top of MCUXpresso SDK peripheral drivers and leverage native RT-OS services to better comply to the RTOS cases.
- Real time operation systems (RTOS) for FreeRTOS OS.
- Stacks and middleware in source or object formats including:
  - CMSIS-DSP, a suite of common signal processing functions.
  - The MCUXpresso SDK comes complete with software examples demonstrating the usage of the peripheral drivers, RTOS wrapper drivers, middleware, and RTOSes.

  All demo applications and driver examples are provided with projects for the following toolchains:
  - IAR Embedded Workbench
  - GNU Arm Embedded Toolchain

The peripheral drivers and RTOS driver wrappers can be used across multiple devices within the product family without modification. The configuration items for each driver are encapsulated into C language data structures. Device-specific configuration information is provided as part of the MCUXpresso SDK and need not be modified by the user. If necessary, the user is able to modify the peripheral driver and RTOS wrapper driver configuration during runtime. The driver examples demonstrate how to configure the drivers by passing the proper configuration data to the APIs. The folder structure is organized to reduce the total number of includes required to compile a project.

The rest of this document describes the API references in detail for the peripheral drivers and RT-OS wrapper drivers. For the latest version of this and other MCUXpresso SDK documents, see the `mcuxpresso.nxp.com/apidoc/`.

| Deliverable | Location |
|---|---|
| Demo Applications | <install_dir>/boards/<board_name>/demo_-apps |
| Driver Examples | <install_dir>/boards/<board_name>/driver_-examples |
| Documentation | <install_dir>/docs |
| Middleware | <install_dir>/middleware |
| Drivers | <install_dir>/<device_name>/drivers/ |
| CMSIS Standard Arm Cortex-M Headers, math and DSP Libraries | <install_dir>/CMSIS |
| Device Startup and Linker | <install_dir>/<device_name>/<toolchain>/ |
| MCUXpresso SDK Utilities | <install_dir>/devices/<device_name>/utilities |
| RTOS Kernel Code | <install_dir>/rtos |

**MCUXpresso SDK Folder Structure**

# Chapter 2
# Trademarks

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

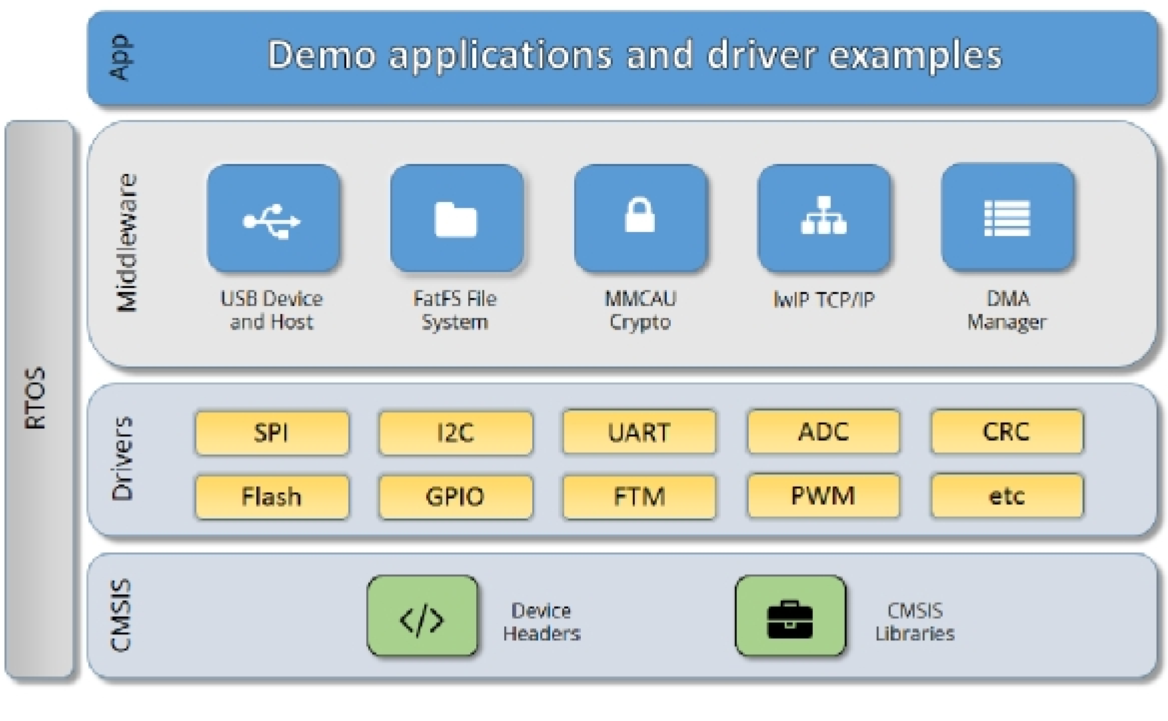
# Chapter 3
# Architectural Overview

This chapter provides the architectural overview for the MCUXpresso Software Development Kit (MCU-Xpresso SDK). It describes each layer within the architecture and its associated components.

**Overview**

The MCUXpresso SDK architecture consists of five key components listed below.

1. The Arm Cortex Microcontroller Software Interface Standard (CMSIS) CORE compliance device-specific header files, SOC Header, and CMSIS math/DSP libraries.
2. Peripheral Drivers
3. Real-time Operating Systems (RTOS)
4. Stacks and Middleware that integrate with the MCUXpresso SDK
5. Demo Applications based on the MCUXpresso SDK




**MCUXpresso SDK Block Diagram**

**MCU header files**

Each supported MCU device in the MCUXpresso SDK has an overall System-on Chip (SoC) memory-mapped header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers. The overall SoC header file provides access to the peripheral registers through pointers and predefined bit masks. In addition to the overall SoC memory-mapped header file, the MCUXpresso SDK includes a feature header file for each device. The feature header file allows NXP to deliver a single software driver for a given peripheral. The feature file ensures that the driver is properly compiled for the target SOC.

**CMSIS Support**

Along with the SoC header files and peripheral extension header files, the MCUXpresso SDK also includes common CMSIS header files for the Arm Cortex-M core and the math and DSP libraries from the latest CMSIS release. The CMSIS DSP library source code is also included for reference.

**MCUXpresso SDK Peripheral Drivers**

The MCUXpresso SDK peripheral drivers mainly consist of low-level functional APIs for the MCU product family on-chip peripherals and also of high-level transactional APIs for some bus drivers/DMA driver/eDMA driver to quickly enable the peripherals and perform transfers.

All MCUXpresso SDK peripheral drivers only depend on the CMSIS headers, device feature files, fsl_common.h, and fsl_clock.h files so that users can easily pull selected drivers and their dependencies into projects. With the exception of the clock/power-relevant peripherals, each peripheral has its own driver. Peripheral drivers handle the peripheral clock gating/ungating inside the drivers during initialization and deinitialization respectively.

Low-level functional APIs provide common peripheral functionality, abstracting the hardware peripheral register accesses into a set of stateless basic functional operations. These APIs primarily focus on the control, configuration, and function of basic peripheral operations. The APIs hide the register access details and various MCU peripheral instantiation differences so that the application can be abstracted from the low-level hardware details. The API prototypes are intentionally similar to help ensure easy portability across supported MCUXpresso SDK devices.

Transactional APIs provide a quick method for customers to utilize higher-level functionality of the peripherals. The transactional APIs utilize interrupts and perform asynchronous operations without user intervention. Transactional APIs operate on high-level logic that requires data storage for internal operation context handling. However, the Peripheral Drivers do not allocate this memory space. Rather, the user passes in the memory to the driver for internal driver operation. Transactional APIs ensure the NVIC is enabled properly inside the drivers. The transactional APIs do not meet all customer needs, but provide a baseline for development of custom user APIs.

Note that the transactional drivers never disable an NVIC after use. This is due to the shared nature of interrupt vectors on devices. It is up to the user to ensure that NVIC interrupts are properly disabled after usage is complete.

**Interrupt handling for transactional APIs**

A double weak mechanism is introduced for drivers with transactional API. The double weak indicates two levels of weak vector entries. See the examples below:

```
        PUBWEAK SPI0_IRQHandler
        PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

```
LDR     R0, =SPI0_DriverIRQHandler
BX      R0
```

The first level of the weak implementation are the functions defined in the vector table. In the devices/<D-EVICE_NAME>/<TOOLCHAIN>/startup_<DEVICE_NAME>.s/.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0_DriverIRQHandler) jumps to itself (B). The MCUXpresso SDK drivers with transactional APIs provide the reimplementation of the second layer function inside of the peripheral driver. If the MCUXpresso SDK drivers with transactional APIs are linked into the image, the SPI0_-DriverIRQHandler is replaced with the function implemented in the MCUXpresso SDK SPI driver.

The reason for implementing the double weak functions is to provide a better user experience when using the transactional APIs. For drivers with a transactional function, call the transactional APIs and the drivers complete the interrupt-driven flow. Users are not required to redefine the vector entries out of the box. At the same time, if users are not satisfied by the second layer weak function implemented in the MCU-Xpresso SDK drivers, users can redefine the first layer weak function and implement their own interrupt handler functions to suit their implementation.

The limitation of the double weak mechanism is that it cannot be used for peripherals that share the same vector entry. For this use case, redefine the first layer weak function to enable the desired peripheral interrupt functionality. For example, if the MCU's UART0 and UART1 share the same vector entry, redefine the UART0_UART1_IRQHandler according to the use case requirements.

**Feature Header Files**

The peripheral drivers are designed to be reusable regardless of the peripheral functional differences from one MCU device to another. An overall Peripheral Feature Header File is provided for the MCUXpresso SDK-supported MCU device to define the features or configuration differences for each sub-family device.

**Application**

See the *Getting Started with MCUXpresso SDK* document (MCUXSDKGSUG).

# Chapter 4
# Clock Driver

## 4.1 Overview

The MCUXpresso SDK provides APIs for MCUXpresso SDK devices' clock operation.

The clock driver supports:

- Clock generator (PLL, FLL, and so on) configuration
- Clock mux and divider configuration
- Getting clock frequency

### Files

- file fsl_clock.h

### Data Structures

- struct sim_clock_config_t
  *SIM configuration structure for clock setting. More...*
- struct osc_config_t
  *OSC Initialization Configuration Structure. More...*
- struct ics_config_t
  *ICS configuration structure. More...*

### Macros

- #define ICS_CONFIG_CHECK_PARAM 0U
  *Configures whether to check a parameter in a function.*
- #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0
  *Configure whether driver controls clock.*
- #define UART_CLOCKS
  *Clock ip name array for UART.*
- #define ADC_CLOCKS
  *Clock ip name array for ADC16.*
- #define IRQ_CLOCKS
  *Clock ip name array for IRQ.*
- #define KBI_CLOCKS
  *Clock ip name array for KBI.*
- #define SPI_CLOCKS
  *Clock ip name array for SPI.*
- #define I2C_CLOCKS
  *Clock ip name array for I2C.*
- #define FTM_CLOCKS
  *Clock ip name array for FTM.*
- #define ACMP_CLOCKS

*Clock ip name array for CMP.*
- #define CRC_CLOCKS
  *Clock ip name array for CRC.*
- #define PWT_CLOCKS
  *Clock ip name array for PWT.*
- #define PIT_CLOCKS
  *Clock ip name array for PIT.*
- #define RTC_CLOCKS
  *Clock ip name array for RTC.*
- #define LPO_CLK_FREQ 1000U
  *LPO clock frequency.*

## Enumerations

- enum clock_name_t {
  kCLOCK_CoreSysClk,
  kCLOCK_PlatClk,
  kCLOCK_BusClk,
  kCLOCK_FlashClk,
  kCLOCK_Osc0ErClk,
  kCLOCK_ICSFixedFreqClk,
  kCLOCK_ICSInternalRefClk,
  kCLOCK_ICSFllClk,
  kCLOCK_ICSOutClk,
  kCLOCK_TimerClk,
  kCLOCK_LpoClk }
  *Clock name used to get clock frequency.*
- enum clock_ip_name_t
  *Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.*
- enum _osc_work_mode {
  kOSC_ModeExt = 0U,
  kOSC_ModeOscLowPower = OSC_CR_OSCOS_MASK,
  kOSC_ModeOscHighGain = OSC_CR_HGO_MASK | OSC_CR_OSCOS_MASK }
  *OSC work mode.*
- enum _osc_enable_mode {
  kOSC_Enable = OSC_CR_OSCEN_MASK,
  kOSC_EnableInStop = OSC_CR_OSCSTEN_MASK }
  *OSC enable mode.*
- enum ics_fll_src_t {
  kICS_FllSrcExternal,
  kICS_FllSrcInternal }
  *ICS FLL reference clock source select.*
- enum ics_clkout_src_t {
  kICS_ClkOutSrcFll,
  kICS_ClkOutSrcInternal,
  kICS_ClkOutSrcExternal }
  *ICSOUT clock source.*
- enum _ics_status {

kStatus_ICS_ModeUnreachable = MAKE_STATUS(kStatusGroup_ICS, 0),
kStatus_ICS_SourceUsed = MAKE_STATUS(kStatusGroup_ICS, 1) }
*ICS status.*
- enum _ics_irclk_enable_mode {
kICS_IrclkDisable = 0U,
kICS_IrclkEnable = ICS_C1_IRCLKEN_MASK,
kICS_IrclkEnableInStop = ICS_C1_IREFSTEN_MASK }
*ICS internal reference clock (ICSIRCLK) enable mode definition.*
- enum ics_mode_t {
kICS_ModeFEI = 0U,
kICS_ModeFBI,
kICS_ModeBILP,
kICS_ModeFEE,
kICS_ModeFBE,
kICS_ModeBELP,
kICS_ModeError }
*ICS mode definitions.*

## Functions

- static void CLOCK_EnableClock (clock_ip_name_t name)
*Enable the clock for specific IP.*
- static void CLOCK_DisableClock (clock_ip_name_t name)
*Disable the clock for specific IP.*
- static void CLOCK_SetOutDiv (uint32_t outdiv1, uint32_t outdiv2, uint32_t outdiv3)
*clock divider*
- uint32_t CLOCK_GetFreq (clock_name_t clockName)
*Gets the clock frequency for a specific clock name.*
- uint32_t CLOCK_GetCoreSysClkFreq (void)
*Get the core clock or system clock frequency.*
- uint32_t CLOCK_GetBusClkFreq (void)
*Get the bus clock frequency.*
- uint32_t CLOCK_GetFlashClkFreq (void)
*Get the flash clock frequency.*
- uint32_t CLOCK_GetOsc0ErClkFreq (void)
*Get the OSC0 external reference clock frequency (OSC0ERCLK).*
- uint32_t CLOCK_GetTimerClkFreq (void)
*Gets the Timer(FTM/PWT) clock frequency.*
- void CLOCK_SetSimConfig (sim_clock_config_t const *config)
*Set the clock configure in SIM module.*
- static void CLOCK_SetSimSafeDivs (void)
*Set the system clock dividers in SIM to safe value.*

## Variables

- volatile uint32_t g_xtal0Freq
*External XTAL0 (OSC0) clock frequency.*

## Driver version

- #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))
  *CLOCK driver version 2.2.1.*

## ICS frequency functions.

- uint32_t CLOCK_GetICSOutClkFreq (void)
  *Gets the ICS output clock (ICSOUTCLK) frequency.*
- uint32_t CLOCK_GetFllFreq (void)
  *Gets the ICS FLL clock (ICSFLLCLK) frequency.*
- uint32_t CLOCK_GetInternalRefClkFreq (void)
  *Gets the ICS internal reference clock (ICSIRCLK) frequency.*
- uint32_t CLOCK_GetICSFixedFreqClkFreq (void)
  *Gets the ICS fixed frequency clock (ICSFFCLK) frequency.*

## ICS clock configuration.

- static void CLOCK_SetLowPowerEnable (bool enable)
  *Enables or disables the ICS low power.*
- static void CLOCK_SetInternalRefClkConfig (uint8_t enableMode)
  *Configures the Internal Reference clock (ICSIRCLK).*
- static void CLOCK_SetFllExtRefDiv (uint8_t rdiv)
  *Set the FLL external reference clock divider value.*

## ICS clock lock monitor functions.

- static void CLOCK_SetOsc0MonitorMode (bool enable)
  *Sets the OSC0 clock monitor mode.*

## OSC configuration

- void CLOCK_InitOsc0 (osc_config_t const ∗config)
  *Initializes the OSC0.*
- void CLOCK_DeinitOsc0 (void)
  *Deinitializes the OSC0.*

## External clock frequency

- static void CLOCK_SetXtal0Freq (uint32_t freq)
  *Sets the XTAL0 frequency based on board settings.*
- static void CLOCK_SetOsc0Enable (uint8_t enable)
  *Sets the OSC enable.*

## ICS mode functions.

- ics_mode_t CLOCK_GetMode (void)
  *Gets the current ICS mode.*
- status_t CLOCK_SetFeiMode (uint8_t bDiv)
  *Sets the ICS to FEI mode.*

- status_t CLOCK_SetFeeMode (uint8_t bDiv, uint8_t rDiv)
  *Sets the ICS to FEE mode.*
- status_t CLOCK_SetFbiMode (uint8_t bDiv)
  *Sets the ICS to FBI mode.*
- status_t CLOCK_SetFbeMode (uint8_t bDiv, uint8_t rDiv)
  *Sets the ICS to FBE mode.*
- status_t CLOCK_SetBilpMode (uint8_t bDiv)
  *Sets the ICS to BILP mode.*
- status_t CLOCK_SetBelpMode (uint8_t bDiv)
  *Sets the ICS to BELP mode.*
- status_t CLOCK_BootToFeiMode (uint8_t bDiv)
  *Sets the ICS to FEI mode during system boot up.*
- status_t CLOCK_BootToFeeMode (uint8_t bDiv, uint8_t rDiv)
  *Sets the ICS to FEE mode during system bootup.*
- status_t CLOCK_BootToBilpMode (uint8_t bDiv)
  *Sets the ICS to BILP mode during system boot up.*
- status_t CLOCK_BootToBelpMode (uint8_t bDiv)
  *Sets the ICS to BELP mode during system boot up.*
- status_t CLOCK_SetIcsConfig (ics_config_t const ∗config)
  *Sets the ICS to a target mode.*

## 4.2 Data Structure Documentation

### 4.2.1 struct sim_clock_config_t

#### Data Fields

- uint8_t outDiv1
  *OUTDIV1.*
- uint8_t outDiv2
  *OUTDIV2.*
- uint8_t outDiv3
  *OUTDIV3.*
- uint8_t busClkPrescaler
  *A option prescaler for bus clock.*

### 4.2.2 struct osc_config_t

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

1. freq: The external frequency.
2. workMode: The OSC module mode.
3. enableMode: The OSC enable mode.

## Data Fields

- uint32_t freq
  *External clock frequency.*
- uint8_t workMode
  *OSC work mode setting.*
- uint8_t enableMode
  *Configuration for OSCERCLK.*

### Field Documentation

#### (1) uint32_t osc_config_t::freq

#### (2) uint8_t osc_config_t::workMode

#### (3) uint8_t osc_config_t::enableMode

## 4.2.3 struct ics_config_t

When porting to a new board, set the following members according to the board setting:

1. icsMode: ICS mode
2. irClkEnableMode: ICSIRCLK enable mode
3. rDiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by rDiv is in the 31.25 kHz to 39.0625 kHz range.
4. bDiv, this divider determine the ISCOUT clock

## Data Fields

- ics_mode_t icsMode
  *ICS mode.*
- uint8_t irClkEnableMode
  *ICSIRCLK enable mode.*
- uint8_t rDiv
  *Divider for external reference clock, ICS_C1[RDIV].*
- uint8_t bDiv
  *Divider for ICS output clock ICS_C2[BDIV].*

### Field Documentation

#### (1) ics_mode_t ics_config_t::icsMode

#### (2) uint8_t ics_config_t::irClkEnableMode

#### (3) uint8_t ics_config_t::rDiv

#### (4) uint8_t ics_config_t::bDiv

# 4.3 Macro Definition Documentation

### 4.3.1 #define ICS_CONFIG_CHECK_PARAM 0U

Some ICS settings must be changed with conditions, for example:

1. ICSIRCLK settings, such as the source, divider, and the trim value should not change when ICSIRCLK is used as a system clock source.
2. ICS_C7[OSCSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BELP/PBE modes.
3. The users should only switch between the supported clock modes.

ICS functions check the parameter and ICS status before setting, if not allowed to change, the functions return error. The parameter checking increases code size, if code size is a critical requirement, change ICS_CONFIG_CHECK_PARAM to 0 to disable parameter checking.

### 4.3.2 #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0

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

Note

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

### 4.3.3 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))

### 4.3.4 #define UART_CLOCKS

**Value:**

```
{                    \
        kCLOCK_Uart0 \
    }
```

### 4.3.5 #define ADC_CLOCKS

**Value:**

```
{                   \
        kCLOCK_Adc0 \
    }
```

## 4.3.6 #define IRQ_CLOCKS

**Value:**

```
{               \
        kCLOCK_Irq0 \
    }
```

## 4.3.7 #define KBI_CLOCKS

**Value:**

```
{                            \
        kCLOCK_Kbi0, kCLOCK_Kbi1 \
    }
```

## 4.3.8 #define SPI_CLOCKS

**Value:**

```
{               \
        kCLOCK_Spi0 \
    }
```

## 4.3.9 #define I2C_CLOCKS

**Value:**

```
{               \
        kCLOCK_I2c0 \
    }
```

## 4.3.10 #define FTM_CLOCKS

**Value:**

```
{                                            \
        kCLOCK_Ftm0, kCLOCK_IpInvalid, kCLOCK_Ftm2 \
    }
```

### 4.3.11 #define ACMP_CLOCKS

**Value:**

```
{                              \
        kCLOCK_Acmp0, kCLOCK_Acmp1 \
    }
```

### 4.3.12 #define CRC_CLOCKS

**Value:**

```
{                \
        kCLOCK_Crc0, \
    }
```

### 4.3.13 #define PWT_CLOCKS

**Value:**

```
{                \
        kCLOCK_Pwt0, \
    }
```

### 4.3.14 #define PIT_CLOCKS

**Value:**

```
{                \
        kCLOCK_Pit0, \
    }
```

### 4.3.15 #define RTC_CLOCKS

**Value:**

```
{                \
        kCLOCK_Rtc0, \
    }
```

## 4.4 Enumeration Type Documentation

### 4.4.1 enum clock_name_t

Enumerator

***kCLOCK_CoreSysClk*** Core/system clock.
***kCLOCK_PlatClk*** Platform clock.
***kCLOCK_BusClk*** Bus clock.
***kCLOCK_FlashClk*** Flash clock.
***kCLOCK_Osc0ErClk*** OSC0 external reference clock (OSC0ERCLK)
***kCLOCK_ICSFixedFreqClk*** ICS fixed frequency clock (ICSFFCLK)
***kCLOCK_ICSInternalRefClk*** ICS internal reference clock (ICSIRCLK)
***kCLOCK_ICSFllClk*** ICSFLLCLK.
***kCLOCK_ICSOutClk*** ICS Output clock.
***kCLOCK_TimerClk*** TIMER clock for FTM and PWT.
***kCLOCK_LpoClk*** LPO clock.

### 4.4.2 enum clock_ip_name_t

### 4.4.3 enum _osc_work_mode

Enumerator

***kOSC_ModeExt*** OSC source from external clock.
***kOSC_ModeOscLowPower*** Oscillator low freq low power.
***kOSC_ModeOscHighGain*** Oscillator low freq high gain.

### 4.4.4 enum _osc_enable_mode

Enumerator

***kOSC_Enable*** Enable.
***kOSC_EnableInStop*** Enable in stop mode.

### 4.4.5 enum ics_fll_src_t

Enumerator

***kICS_FllSrcExternal*** External reference clock is selected.
***kICS_FllSrcInternal*** The slow internal reference clock is selected.

### 4.4.6 enum ics_clkout_src_t

Enumerator

***kICS_ClkOutSrcFll*** Output of the FLL is selected (reset default)
***kICS_ClkOutSrcInternal*** Internal reference clock is selected, FLL is bypassed.
***kICS_ClkOutSrcExternal*** External reference clock is selected, FLL is bypassed.

### 4.4.7 enum _ics_status

Enumerator

***kStatus_ICS_ModeUnreachable*** Can't switch to target mode.
***kStatus_ICS_SourceUsed*** Can't change the clock source because it is in use.

### 4.4.8 enum _ics_irclk_enable_mode

Enumerator

***kICS_IrclkDisable*** ICSIRCLK disable.
***kICS_IrclkEnable*** ICSIRCLK enable.
***kICS_IrclkEnableInStop*** ICSIRCLK enable in stop mode.

### 4.4.9 enum ics_mode_t

Enumerator

***kICS_ModeFEI*** FEI - FLL Engaged Internal.
***kICS_ModeFBI*** FBI - FLL Bypassed Internal.
***kICS_ModeBILP*** BILP - Bypassed Low Power Internal.
***kICS_ModeFEE*** FEE - FLL Engaged External.
***kICS_ModeFBE*** FBE - FLL Bypassed External.
***kICS_ModeBELP*** BELP - Bypassed Low Power External.
***kICS_ModeError*** Unknown mode.

## 4.5 Function Documentation

### 4.5.1 static void CLOCK_EnableClock ( clock_ip_name_t *name* ) [inline], [static]

Parameters

| | |
|---|---|
| *name* | Which clock to enable, see clock_ip_name_t. |

### 4.5.2 static void CLOCK_DisableClock ( clock_ip_name_t *name* ) [inline], [static]

Parameters

| | |
|---|---|
| *name* | Which clock to disable, see clock_ip_name_t. |

### 4.5.3 static void CLOCK_SetOutDiv ( uint32_t *outdiv1*, uint32_t *outdiv2*, uint32_t *outdiv3* ) [inline], [static]

Set the SIM_CLKDIV[OUTDIV1], SIM_CLKDIV[OUTDIV2], SIM_CLKDIV[OUTDIV3]. Carefully configure the OUTDIV1 and OUTDIV2 to avoid bus clock frequency higher than 24MHZ.

Parameters

| | |
|---|---|
| *outdiv1* | Clock 1 output divider value. |
| *outdiv2* | Clock 2 output divider value. |
| *outdiv3* | Clock 3 output divider value. |

### 4.5.4 uint32_t CLOCK_GetFreq ( clock_name_t *clockName* )

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in clock_name_t. The ICS must be properly configured before using this function.

Parameters

| | |
|---|---|
| *clockName* | Clock names defined in clock_name_t |

Returns

Clock frequency value in Hertz

### 4.5.5 uint32_t CLOCK_GetCoreSysClkFreq ( void )

Returns

Clock frequency in Hz.

### 4.5.6 uint32_t CLOCK_GetBusClkFreq ( void )

Returns

Clock frequency in Hz.

### 4.5.7 uint32_t CLOCK_GetFlashClkFreq ( void )

Returns

Clock frequency in Hz.

### 4.5.8 uint32_t CLOCK_GetOsc0ErClkFreq ( void )

Returns

Clock frequency in Hz.

### 4.5.9 uint32_t CLOCK_GetTimerClkFreq ( void )

This function gets the Timer clock frequency in Hz based on the current ICSOUTCLK.

Returns

The frequency of Timer(FTM/PWT) clock.

### 4.5.10 void CLOCK_SetSimConfig ( sim_clock_config_t const * *config* )

This function sets system layer clock settings in SIM module.

Parameters

| | |
|---|---|
| *config* | Pointer to the configure structure. |

### 4.5.11 static void CLOCK_SetSimSafeDivs ( void ) [inline], [static]

The system level clocks (core clock, bus clock, and flash clock) must be in allowed ranges. During ICS clock mode switch, the ICS output clock changes then the system level clocks may be out of range. This function could be used before ICS mode change, to make sure system level clocks are in allowed range.

### 4.5.12 uint32_t CLOCK_GetICSOutClkFreq ( void )

This function gets the ICS output clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSOUTCLK.

### 4.5.13 uint32_t CLOCK_GetFllFreq ( void )

This function gets the ICS FLL clock frequency in Hz based on the current ICS register value. The FLL is enabled in FEI/FBI/FEE/FBE mode and disabled in low power state in other modes.

Returns

The frequency of ICSFLLCLK.

### 4.5.14 uint32_t CLOCK_GetInternalRefClkFreq ( void )

This function gets the ICS internal reference clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSIRCLK.

### 4.5.15 uint32_t CLOCK_GetICSFixedFreqClkFreq ( void )

This function gets the ICS fixed frequency clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSFFCLK.

### 4.5.16 static void CLOCK_SetLowPowerEnable ( bool *enable* ) [inline], [static]

Enabling the ICS low power disables the PLL and FLL in bypass modes. In other words, in FBE and PBE modes, enabling low power sets the ICS to BELP mode. In FBI and PBI modes, enabling low power sets the ICS to BILP mode. When disabling the ICS low power, the PLL or FLL are enabled based on ICS settings.

Parameters

| | |
|---|---|
| *enable* | True to enable ICS low power, false to disable ICS low power. |

### 4.5.17 static void CLOCK_SetInternalRefClkConfig ( uint8_t *enableMode* ) [inline], [static]

This function sets the ICSIRCLK base on parameters. This function also sets whether the `ICSIRCLK` is enabled in stop mode.

Parameters

| | |
|---|---|
| *enableMode* | ICSIRCLK enable mode, OR'ed value of _ICS_irclk_enable_mode. |

Return values

| | |
|---|---|
| *kStatus_ICS_SourceUsed* | Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs. |
| *kStatus_Success* | ICSIRCLK configuration finished successfully. |

### 4.5.18 static void CLOCK_SetFllExtRefDiv ( uint8_t *rdiv* ) [inline], [static]

Sets the FLL external reference clock divider value, the register ICS_C1[RDIV]. Resulting frequency must be in the range 31.25KHZ to 39.0625KHZ.

Parameters

| | |
|---|---|
| *rdiv* | The FLL external reference clock divider value, ICS_C1[RDIV]. |

### 4.5.19 static void CLOCK_SetOsc0MonitorMode ( bool *enable* ) [inline], [static]

This function sets the OSC0 clock monitor mode. See ics_monitor_mode_t for details.

Parameters

| | |
|---|---|
| *enable* | true to enable clock monitor, false to disable clock monitor. |

### 4.5.20 void CLOCK_InitOsc0 ( osc_config_t const * *config* )

This function initializes the OSC0 according to the board configuration.

Parameters

| | |
|---|---|
| *config* | Pointer to the OSC0 configuration structure. |

### 4.5.21 void CLOCK_DeinitOsc0 ( void )

This function deinitializes the OSC0.

### 4.5.22 static void CLOCK_SetXtal0Freq ( uint32_t *freq* ) [inline], [static]

Parameters

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

### 4.5.23 static void CLOCK_SetOsc0Enable ( uint8_t *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *enable* | osc enable mode. |

### 4.5.24 ics_mode_t CLOCK_GetMode ( void )

This function checks the ICS registers and determines the current ICS mode.

Returns

Current ICS mode or error code; See ics_mode_t.

### 4.5.25 status_t CLOCK_SetFeiMode ( uint8_t *bDiv* )

This function sets the ICS to FEI mode. If setting to FEI mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.26 status_t CLOCK_SetFeeMode ( uint8_t *bDiv,* uint8_t *rDiv* )

This function sets the ICS to FEE mode. If setting to FEE mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |

| | |
|---|---|
| *rDiv* | FLL reference clock divider setting, RDIV. |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.27 status_t CLOCK_SetFbiMode ( uint8_t *bDiv* )

This function sets the ICS to FBI mode. If setting to FBI mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.28 status_t CLOCK_SetFbeMode ( uint8_t *bDiv,* uint8_t *rDiv* )

This function sets the ICS to FBE mode. If setting to FBE mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |
| *rDiv* | FLL reference clock divider setting, RDIV. |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |

| | |
|---|---|
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.29 status_t CLOCK_SetBilpMode ( uint8_t *bDiv* )

This function sets the ICS to BILP mode. If setting to BILP mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.30 status_t CLOCK_SetBelpMode ( uint8_t *bDiv* )

This function sets the ICS to BELP mode. If setting to BELP mode fails from the current mode, this function returns an error.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.31 status_t CLOCK_BootToFeiMode ( uint8_t *bDiv* )

This function sets the ICS to FEI mode from the reset mode. It can also be used to set up ICS during system boot up.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider. |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.32 status_t CLOCK_BootToFeeMode ( uint8_t *bDiv*, uint8_t *rDiv* )

This function sets ICS to FEE mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider. |
| *rDiv* | FLL reference clock divider setting, RDIV. |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.33 status_t CLOCK_BootToBilpMode ( uint8_t *bDiv* )

This function sets the ICS to BILP mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider. |

Return values

| | |
|---|---|
| *kStatus_ICS_SourceUsed* | Could not change ICSIRCLK setting. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.34 status_t CLOCK_BootToBelpMode ( uint8_t *bDiv* )

This function sets the ICS to BELP mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

| | |
|---|---|
| *bDiv* | bus clock divider. |

Return values

| | |
|---|---|
| *kStatus_ICS_Mode-Unreachable* | Could not switch to the target mode. |
| *kStatus_Success* | Switched to the target mode successfully. |

### 4.5.35 status_t CLOCK_SetIcsConfig ( ics_config_t const * *config* )

This function sets ICS to a target mode defined by the configuration structure. If switching to the target mode fails, this function chooses the correct path.

Parameters

| | |
|---|---|
| *config* | Pointer to the target ICS mode configuration structure. |

Returns

Return kStatus_Success if switched successfully; Otherwise, it returns an error code _ICS_status.

Note

If the external clock is used in the target mode, ensure that it is enabled. For example, if the OSC0 is used, set up OSC0 correctly before calling this function.

## 4.6 Variable Documentation

### 4.6.1 volatile uint32_t g_xtal0Freq

The XTAL0/EXTAL0 (OSC0) clock frequency in Hz. When the clock is set up, use the function CLOCK_SetXtal0Freq to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
* CLOCK_InitOsc0(...);
* CLOCK_SetXtal0Freq(80000000);
*
```

This is important for the multicore platforms where only one core needs to set up the OSC0 using the CLOCK_InitOsc0. All other cores need to call the CLOCK_SetXtal0Freq to get a valid clock frequency.

# Chapter 5
# PORT Driver

## 5.1 Overview

This driver configures the PORT, including function mux, filter, pull up or down, and so on.

### Macros

- #define FSL_PORT_FILTER_SELECT_BITMASK (0x3U)
  *The IOFLT Filter selection bit mask .*

### Enumerations

- enum port_module_t {
  kPORT_NMI = SIM_SOPT_NMIE_MASK,
  kPORT_RESET = SIM_SOPT_RSTPE_MASK,
  kPORT_SWDE = SIM_SOPT_SWDE_MASK,
  kPORT_I2C0 = SIM_PINSEL_I2C0PS_MASK,
  kPORT_SPI0 = SIM_PINSEL_SPI0PS_MASK,
  kPORT_UART0 = SIM_PINSEL_UART0PS_MASK,
  kPORT_FTM0CH0 = SIM_PINSEL_FTM0PS0_MASK,
  kPORT_FTM0CH1 = SIM_PINSEL_FTM0PS1_MASK,
  kPORT_FTM2CH2 = SIM_PINSEL_FTM2PS2_MASK,
  kPORT_FTM2CH3 = SIM_PINSEL_FTM2PS3_MASK,
  kPORT_FTM0CLK = SIM_PINSEL_FTM0CLKPS_MASK,
  kPORT_FTM2CLK = SIM_PINSEL_FTM2CLKPS_MASK,
  kPORT_PWTCLK = (int)SIM_PINSEL_PWTCLKPS_MASK }
  *Module or peripheral for port pin selection.*
- enum port_type_t {
  kPORT_PTA = 0U,
  kPORT_PTB = 1U,
  kPORT_PTC = 2U }
  *Port type.*
- enum port_pin_index_t {
  kPORT_PinIdx0 = 0U,
  kPORT_PinIdx1 = 1U,
  kPORT_PinIdx2 = 2U,
  kPORT_PinIdx3 = 3U,
  kPORT_PinIdx4 = 4U,
  kPORT_PinIdx5 = 5U,
  kPORT_PinIdx6 = 6U,
  kPORT_PinIdx7 = 7U }

*Pin number, Notice this index enum has been deprecated and it will be removed in the next release.*

- enum port_pin_select_t {
  kPORT_NMI_OTHERS = 0U,
  kPORT_NMI_NMIE = 1U,
  kPORT_RST_OTHERS = 0U,
  kPORT_RST_RSTPE = 1U,
  kPORT_SWDE_OTHERS = 0U,
  kPORT_SWDE_SWDE = 1U,
  kPORT_I2C0_SCLPTA3_SDAPTA2 = 0U,
  kPORT_I2C0_SCLPTB7_SDAPTB6 = 1U,
  kPORT_SPI0_SCKPTB2_MOSIPTB3_MISOPTB4_PCSPTB5 = 0U,
  kPORT_SPI0_SCKPTA6_MOSIPTA7_MISOPTB1_PCSPTB0,
  kPORT_UART0_RXPTB0_TXPTB1 = 0U,
  kPORT_UART0_RXPTA2_TXPTA3 = 1U,
  kPORT_FTM0_CH0_PTA0 = 0U,
  kPORT_FTM0_CH0_PTB2 = 1U,
  kPORT_FTM0_CH1_PTA1 = 0U,
  kPORT_FTM0_CH1_PTB3 = 1U,
  kPORT_FTM2_CH2_PTC2 = 0U,
  kPORT_FTM2_CH2_PTC4 = 1U,
  kPORT_FTM2_CH3_PTC3 = 0U,
  kPORT_FTM2_CH3_PTC5 = 1U,
  kPORT_FTM0CLK_TCLK1 = 0U,
  kPORT_FTM0CLK_TCLK2 = 1U,
  kPORT_FTM2CLK_TCLK1 = 0U,
  kPORT_FTM2CLK_TCLK2 = 1U,
  kPORT_PWTCLK_TCLK1 = 0U,
  kPORT_PWTCLK_TCLK2 = 1U }

  *Pin selection.*
- enum port_filter_pin_t {
  kPORT_FilterPTA = PORT_IOFLT_FLTA_SHIFT,
  kPORT_FilterPTB = PORT_IOFLT_FLTB_SHIFT,
  kPORT_FilterPTC = PORT_IOFLT_FLTC_SHIFT,
  kPORT_FilterIIC = PORT_IOFLT_FLTIIC_SHIFT,
  kPORT_FilterFTM0 = PORT_IOFLT_FLTFTM0_SHIFT,
  kPORT_FilterPWT = PORT_IOFLT_FLTPWT_SHIFT,
  kPORT_FilterRST = PORT_IOFLT_FLTRST_SHIFT,
  kPORT_FilterKBI0 = PORT_IOFLT_FLTKBI0_SHIFT,
  kPORT_FilterKBI1 = PORT_IOFLT_FLTKBI1_SHIFT,
  kPORT_FilterNMI = PORT_IOFLT_FLTNMI_SHIFT }

  *The PORT pins for input glitch filter configure.*
- enum port_filter_select_t {
  kPORT_BUSCLK_OR_NOFILTER,
  kPORT_FILTERDIV1 = 1U,
  kPORT_FILTERDIV2 = 2U,

kPORT_FILTERDIV3_OR_BUSCLK = 3U }
*The Filter selection for input pins.*
- enum port_highdrive_pin_t {
kPORT_HighDrive_PTB5 = PORT_HDRVE_PTB5_MASK,
kPORT_HighDrive_PTC1 = PORT_HDRVE_PTC1_MASK,
kPORT_HighDrive_PTC5 = PORT_HDRVE_PTC5_MASK }
*Port pin for high driver enable/disable control.*

### Driver version

- #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
*Version 2.0.2.*

### Configuration

- void PORT_SetPinSelect (port_module_t module, port_pin_select_t pin)
*Selects pin for modules.*
- static void PORT_SetFilterSelect (PORT_Type ∗base, port_filter_pin_t port, port_filter_select_t filter)
*Selects the glitch filter for input pins.*
- static void PORT_SetFilterDIV1WidthThreshold (PORT_Type ∗base, uint8_t threshold)
*Sets the width threshold for glitch filter division set 1.*
- static void PORT_SetFilterDIV2WidthThreshold (PORT_Type ∗base, uint8_t threshold)
*Sets the width threshold for glitch filter division set 2.*
- static void PORT_SetFilterDIV3WidthThreshold (PORT_Type ∗base, uint8_t threshold)
*Sets the width threshold for glitch filter division set 3.*
- static void PORT_SetPinPullUpEnable (PORT_Type ∗base, port_type_t port, uint8_t num, bool enable)
*Enables or disables the port pull up.*
- static void PORT_SetHighDriveEnable (PORT_Type ∗base, port_highdrive_pin_t pin, bool enable)
*Set High drive for port pins.*

## 5.2 Macro Definition Documentation

### 5.2.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

### 5.2.2 #define FSL_PORT_FILTER_SELECT_BITMASK (0x3U)

## 5.3 Enumeration Type Documentation

### 5.3.1 enum port_module_t

Enumerator

***kPORT_NMI*** NMI port pin select.
***kPORT_RESET*** RESET pin select.
***kPORT_SWDE*** Single wire debug port pin.
***kPORT_I2C0*** I2C0 Port pin select.

***kPORT_SPI0*** SPI0 port pin select.
***kPORT_UART0*** UART0 port pin select.
***kPORT_FTM0CH0*** FTM0_CH0 port pin select.
***kPORT_FTM0CH1*** FTM0_CH1 port pin select.
***kPORT_FTM2CH2*** FTM2_CH2 port pin select.
***kPORT_FTM2CH3*** FTM2_CH3 port pin select.
***kPORT_FTM0CLK*** FTM0 clock pin select.
***kPORT_FTM2CLK*** FTM2 clock pin select.
***kPORT_PWTCLK*** PWT clock pin select.

### 5.3.2 enum port_type_t

Enumerator

***kPORT_PTA*** PORT PTA.
***kPORT_PTB*** PORT PTB.
***kPORT_PTC*** PORT PTC.

### 5.3.3 enum port_pin_index_t

Enumerator

***kPORT_PinIdx0*** PORT PIN index 0.
***kPORT_PinIdx1*** PORT PIN index 1.
***kPORT_PinIdx2*** PORT PIN index 2.
***kPORT_PinIdx3*** PORT PIN index 3.
***kPORT_PinIdx4*** PORT PIN index 4.
***kPORT_PinIdx5*** PORT PIN index 5.
***kPORT_PinIdx6*** PORT PIN index 6.
***kPORT_PinIdx7*** PORT PIN index 7.

### 5.3.4 enum port_pin_select_t

Enumerator

***kPORT_NMI_OTHERS*** PTB4/FTM2_CH4 etc function as PTB4/FTM2_CH4 etc.
***kPORT_NMI_NMIE*** PTB4/FTM2_CH4 etc function as NMI.
***kPORT_RST_OTHERS*** PTA5/IRQ etc function as PTA5/IRQ etc.
***kPORT_RST_RSTPE*** PTA5/IRQ etc function as REST.
***kPORT_SWDE_OTHERS*** PTA4/ACMP0 etc function as PTA4/ACMP0 etc.
***kPORT_SWDE_SWDE*** PTA4/ACMP0 etc function as SWD.

***kPORT_I2C0_SCLPTA3_SDAPTA2*** I2C0_SCL and I2C0_SDA are mapped on PTA3 and PTA2, respectively.
***kPORT_I2C0_SCLPTB7_SDAPTB6*** I2C0_SCL and I2C0_SDA are mapped on PTB7 and PTB6, respectively.
***kPORT_SPI0_SCKPTB2_MOSIPTB3_MISOPTB4_PCSPTB5*** SPI0_SCK/MOSI/MISO/PCS0 are mapped on PTB2/PTB3/PTB4/PTB5.
***kPORT_SPI0_SCKPTA6_MOSIPTA7_MISOPTB1_PCSPTB0*** SPI0_SCK/MOSI/MISO/PCS0 are mapped on PTA6/PTA7/PTB1/PTB0.
***kPORT_UART0_RXPTB0_TXPTB1*** UART0_RX and UART0_TX are mapped on PTB0 and PT-B1.
***kPORT_UART0_RXPTA2_TXPTA3*** UART0_RX and UART0_TX are mapped on PTA2 and PT-A3.
***kPORT_FTM0_CH0_PTA0*** FTM0_CH0 channels are mapped on PTA0.
***kPORT_FTM0_CH0_PTB2*** FTM0_CH0 channels are mapped on PTB2.
***kPORT_FTM0_CH1_PTA1*** FTM0_CH1 channels are mapped on PTA1.
***kPORT_FTM0_CH1_PTB3*** FTM0_CH1 channels are mapped on PTB3.
***kPORT_FTM2_CH2_PTC2*** FTM2_CH2 channels are mapped on PTC2.
***kPORT_FTM2_CH2_PTC4*** FTM2_CH2 channels are mapped on PTD2.
***kPORT_FTM2_CH3_PTC3*** FTM2_CH3 channels are mapped on PTC3.
***kPORT_FTM2_CH3_PTC5*** FTM2_CH3 channels are mapped on PTC5.
***kPORT_FTM0CLK_TCLK1*** FTM0 CLK using the TCLK1 pin.
***kPORT_FTM0CLK_TCLK2*** FTM0 CLK using the TCLK2 pin.
***kPORT_FTM2CLK_TCLK1*** FTM2 CLK using the TCLK1 pin.
***kPORT_FTM2CLK_TCLK2*** FTM2 CLK using the TCLK2 pin.
***kPORT_PWTCLK_TCLK1*** PWT CLK using the TCLK1 pin.
***kPORT_PWTCLK_TCLK2*** PWT CLK using the TCLK2 pin.

### 5.3.5 enum port_filter_pin_t

Enumerator

***kPORT_FilterPTA*** Filter for input from PTA.
***kPORT_FilterPTB*** Filter for input from PTB.
***kPORT_FilterPTC*** Filter for input from PTC.
***kPORT_FilterIIC*** Filter for input from I2C.
***kPORT_FilterFTM0*** Filter for input from FTM0.
***kPORT_FilterPWT*** Filter for input from PWT.
***kPORT_FilterRST*** Filter for input from RESET/IRQ.
***kPORT_FilterKBI0*** Filter for input from KBI0.
***kPORT_FilterKBI1*** Filter for input from KBI1.
***kPORT_FilterNMI*** Filter for input from NMI.

### 5.3.6 enum port_filter_select_t

Enumerator

***kPORT_BUSCLK_OR_NOFILTER*** Filter section BUSCLK for PTA~PTC, No filter for REST/-KBI0/KBI1/NMI/PWT/FTM0/I2C.
***kPORT_FILTERDIV1*** Filter Division Set 1.
***kPORT_FILTERDIV2*** Filter Division Set 2.
***kPORT_FILTERDIV3_OR_BUSCLK*** Filter Division Set 3.

### 5.3.7 enum port_highdrive_pin_t

Enumerator

***kPORT_HighDrive_PTB5*** PTB5.
***kPORT_HighDrive_PTC1*** PTC1.
***kPORT_HighDrive_PTC5*** PTC5.

## 5.4 Function Documentation

### 5.4.1 void PORT_SetPinSelect ( port_module_t *module*, port_pin_select_t *pin* )

This API is used to select the port pin for the module with multiple port pin selection. For example the FTM Channel 0 can be mapped to ether PTA0 or PTB2. Select FTM channel 0 map to PTA0 port pin as:

```
* PORT_SetPinSelect(kPORT_FTM0CH0,
     kPORT_FTM0_CH0_PTA0);
*
```

Note

This API doesn't support to select specified ALT for a given port pin. The ALT feature is automatically selected by hardware according to the ALT priority: Low -----> high: Alt1, Alt2, ... when peripheral modules has been enabled.

If you want to select a specified ALT for a given port pin, please add two more steps after calling PORT-_SetPinSelect:

1. Enable module or the port control in the module for the ALT you want to select. For I2C ALT feature:all port enable is controlled by the module enable, so set IICEN in I2CX_C1 to enable the port pins for I2C feature. For KBI ALT feature:each port pin is controlled independently by each bit in KBIx_PE. set related bit in this register to enable the KBI feature in the port pin.
2. Make sure there is no module enabled with higher priority than the ALT module feature you want to select.

Parameters

| | |
|---|---|
| *module* | Modules for pin selection. For NMI/RST module are write-once attribute after reset. |
| *pin* | Port pin selection for modules. |

### 5.4.2 static void PORT_SetFilterSelect ( PORT_Type ∗ *base*, port_filter_pin_t *port*, port_filter_select_t *filter* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |
| *port* | PORT pin, see "port_filter_pin_t". |
| *filter* | Filter select, see "port_filter_select_t". |

### 5.4.3 static void PORT_SetFilterDIV1WidthThreshold ( PORT_Type ∗ *base*, uint8_t *threshold* ) [inline], [static]

‘

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |
| *threshold* | PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - LPOCLK 1 - LPOCLK/2 2 - LPOCLK/4 3 - LPOCLK/8 4 - LPOCLK/16 5 - LPOCLK/32 6 - LPOCLK/64 7 - LPOCLK/128 |

### 5.4.4 static void PORT_SetFilterDIV2WidthThreshold ( PORT_Type ∗ *base*, uint8_t *threshold* ) [inline], [static]

‘

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |

| | |
|---|---|
| *threshold* | PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - BUSCLK/32 1 - BUSCLK/64 2 - BUSCLK/128 3 - BUSCLK/256 4 - BUSCLK/512 5 - BUSCLK/1024 6 - BUSCLK/2048 7 - BUSCLK/4096 |

### 5.4.5 static void PORT_SetFilterDIV3WidthThreshold ( PORT_Type * *base*, uint8_t *threshold* ) [inline], [static]

‘

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |
| *threshold* | PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - BUSCLK/2 1 - BUSCLK/4 2 - BUSCLK/8 3 - BUSCLK/16 |

### 5.4.6 static void PORT_SetPinPullUpEnable ( PORT_Type * *base*, port_type_t *port*, uint8_t *num*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |
| *port* | PORT type, such as PTA/PTB/PTC etc, see "port_type_t". |
| *num* | PORT pin number, such as 0, 1, 2.... For PTC, there are only six pins from 0 $\sim$ 5, the PTC6, PTC7 are not exists in this device. so when set PTC please don't set the 6 and 7, take refer to the reference manual. |
| *enable* | Enable or disable the pull up feature switch. |

### 5.4.7 static void PORT_SetHighDriveEnable ( PORT_Type * *base*, port_highdrive_pin_t *pin*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PORT peripheral base pointer. |

| | |
|---|---|
| *pin* | PORT pin support high drive. |
| *enable* | Enable or disable the high driver feature switch. |

# Chapter 6
# ACMP: Analog Comparator Driver

## 6.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Comparator (ACMP) module of MCUXpresso SDK devices.

## 6.2 Typical use case

### 6.2.1 Normal Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/acmp

### 6.2.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/acmp

## Data Structures

- struct acmp_config_t
  *Configuration for ACMP. More...*
- struct acmp_dac_config_t
  *Configuration for Internal DAC. More...*

## Enumerations

- enum acmp_hysterisis_mode_t {
  kACMP_HysterisisLevel1 = 0U,
  kACMP_HysterisisLevel2 = 1U }
  *Analog Comparator Hysterisis Selection.*
- enum acmp_reference_voltage_source_t {
  kACMP_VrefSourceVin1 = 0U,
  kACMP_VrefSourceVin2 = 1U }
  *DAC Voltage Reference source.*
- enum acmp_interrupt_mode_t {
  kACMP_OutputFallingInterruptMode = 0U,
  kACMP_OutputRisingInterruptMode = 1U,
  kACMP_OutputBothEdgeInterruptMode = 3U }
  *The sensitivity modes of the interrupt trigger.*

- enum acmp_input_channel_selection_t {
kACMP_ExternalReference0 = 0U,
kACMP_ExternalReference1 = 1U,
kACMP_ExternalReference2 = 2U,
kACMP_InternalDACOutput = 3U }
  *The ACMP input channel selection.*
- enum _acmp_status_flags {
kACMP_InterruptFlag = ACMP_CS_ACF_MASK,
kACMP_OutputFlag = ACMP_CS_ACO_MASK }
  *The ACMP status flags.*

### Driver version

- #define FSL_ACMP_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 2U))
  *ACMP driver version 2.0.2.*

### Initialization and deinitialization

- void ACMP_Init (ACMP_Type ∗base, const acmp_config_t ∗config)
  *Initialize the ACMP.*
- void ACMP_Deinit (ACMP_Type ∗base)
  *De-Initialize the ACMP.*
- void ACMP_GetDefaultConfig (acmp_config_t ∗config)
  *Gets the default configuration for ACMP.*
- static void ACMP_Enable (ACMP_Type ∗base, bool enable)
  *Enable/Disable the ACMP module.*
- void ACMP_EnableInterrupt (ACMP_Type ∗base, acmp_interrupt_mode_t mode)
  *Enable the ACMP interrupt and determines the sensitivity modes of the interrupt trigger.*
- static void ACMP_DisableInterrupt (ACMP_Type ∗base)
  *Disable the ACMP interrupt.*
- void ACMP_SetChannelConfig (ACMP_Type ∗base, acmp_input_channel_selection_t PositiveInput, acmp_input_channel_selection_t negativeInout)
  *Configure the ACMP positive and negative input channel.*
- void **ACMP_SetDACConfig** (ACMP_Type ∗base, const acmp_dac_config_t ∗config)
- void ACMP_EnableInputPin (ACMP_Type ∗base, uint32_t mask, bool enable)
  *Enable/Disable ACMP input pin.*
- static uint8_t ACMP_GetStatusFlags (ACMP_Type ∗base)
  *Get ACMP status flags.*
- static void ACMP_ClearInterruptFlags (ACMP_Type ∗base)
  *Clear interrupts status flag.*

## 6.3 Data Structure Documentation

### 6.3.1 struct acmp_config_t

### Data Fields

- bool enablePinOut
  *The comparator output is available on the associated pin.*

- acmp_hysterisis_mode_t hysteresisMode
  *Hysteresis mode.*

#### Field Documentation

##### (1) bool acmp_config_t::enablePinOut

##### (2) acmp_hysterisis_mode_t acmp_config_t::hysteresisMode

### 6.3.2 struct acmp_dac_config_t

#### Data Fields

- uint8_t DACValue
  *Value for DAC Output Voltage.*
- acmp_reference_voltage_source_t referenceVoltageSource
  *Supply voltage reference source.*

#### Field Documentation

##### (1) uint8_t acmp_dac_config_t::DACValue

Available range is 0-63.

##### (2) acmp_reference_voltage_source_t acmp_dac_config_t::referenceVoltageSource

## 6.4 Macro Definition Documentation

### 6.4.1 #define FSL_ACMP_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 2U))

## 6.5 Enumeration Type Documentation

### 6.5.1 enum acmp_hysterisis_mode_t

Enumerator

***kACMP_HysterisisLevel1*** ACMP hysterisis is 20mv. >
***kACMP_HysterisisLevel2*** ACMP hysterisis is 30mv. >

### 6.5.2 enum acmp_reference_voltage_source_t

Enumerator

***kACMP_VrefSourceVin1*** The DAC selects Bandgap as the reference.
***kACMP_VrefSourceVin2*** The DAC selects VDDA as the reference.

### 6.5.3 enum acmp_interrupt_mode_t

Enumerator

***kACMP_OutputFallingInterruptMode*** ACMP interrupt on output falling edge. >
***kACMP_OutputRisingInterruptMode*** ACMP interrupt on output rising edge. >
***kACMP_OutputBothEdgeInterruptMode*** ACMP interrupt on output falling or rising edge. >

### 6.5.4 enum acmp_input_channel_selection_t

Enumerator

***kACMP_ExternalReference0*** External reference 0 is selected to as input channel. >
***kACMP_ExternalReference1*** External reference 1 is selected to as input channel. >
***kACMP_ExternalReference2*** External reference 2 is selected to as input channel. >
***kACMP_InternalDACOutput*** Internal DAC putput is selected to as input channel. >

### 6.5.5 enum _acmp_status_flags

Enumerator

***kACMP_InterruptFlag*** ACMP interrupt on output valid edge. >
***kACMP_OutputFlag*** The current value of the analog comparator output. >

## 6.6 Function Documentation

### 6.6.1 void ACMP_Init ( ACMP_Type * *base*, const acmp_config_t * *config* )

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

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |
| *config* | Pointer to ACMP configuration structure. |

### 6.6.2 void ACMP_Deinit ( ACMP_Type * *base* )

Parameters

| | |
|---|---|
| *base* | ACMP peripheral basic address. |

### 6.6.3 void ACMP_GetDefaultConfig ( acmp_config_t * *config* )

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

```
*   config->enablePinOut = false;
*   config->hysteresisMode = kACMP_HysterisisLevel1;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to ACMP configuration structure. |

### 6.6.4 static void ACMP_Enable ( ACMP_Type * *base,* bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |
| *enable* | Switcher to enable/disable ACMP module. |

### 6.6.5 void ACMP_EnableInterrupt ( ACMP_Type * *base,* acmp_interrupt_mode_t *mode* )

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |
| *mode* | Select one interrupt mode to generate interrupt. |

### 6.6.6 static void ACMP_DisableInterrupt ( ACMP_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |

### 6.6.7 void ACMP_SetChannelConfig ( ACMP_Type ∗ *base,* acmp_input_channel_selection_t *PositiveInput,* acmp_input_channel_selection_t *negativeInout* )

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |
| *PositiveInput* | ACMP Positive Input Select. Refer to "acmp_input_channel_selection_t". |
| *negativeInout* | ACMP Negative Input Select. Refer to "acmp_input_channel_selection_t". |

### 6.6.8 void ACMP_EnableInputPin ( ACMP_Type ∗ *base,* uint32_t *mask,* bool *enable* )

```
The API controls if the corresponding ACMP external pin can be driven by an analog input
```

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |
| *mask* | The mask of the pin associated with channel ADx. Valid range is AD0:0x1U ∼ AD3:0x4U. For example: If enable AD0, AD1 and AD2 pins, mask should be set to 0x7U(0x1 \| 0x2 \| 0x4). |
| *enable* | Switcher to enable/disable ACMP module. |

### 6.6.9 static uint8_t ACMP_GetStatusFlags ( ACMP_Type ∗ *base* ) `[inline]`, `[static]`

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |

Returns

Flags' mask if indicated flags are asserted. See "_acmp_status_flags".

### 6.6.10 static void ACMP_ClearInterruptFlags ( ACMP_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | ACMP peripheral base address. |

# Chapter 7
# ADC: 12-bit Analog to Digital Converter Driver

## 7.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 12-bit Analog to Digital Converter (ADC) module of MCUXpresso SDK devices.

## 7.2 Typical use case

### 7.2.1 Interrupt Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/fsl_-adc

### 7.2.2 Polling Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/fsl_-adc

## Data Structures

- struct adc_config_t
  *ADC converter configuration. More...*
- struct adc_hardware_compare_config_t
  *ADC hardware comparison configuration. More...*
- struct adc_fifo_config_t
  *ADC FIFO configuration. More...*
- struct adc_channel_config_t
  *ADC channel conversion configuration. More...*

## Macros

- #define FSL_ADC_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  *ADC driver version.*

## Enumerations

- enum adc_reference_voltage_source_t {
  kADC_ReferenceVoltageSourceAlt0 = 0U,
  kADC_ReferenceVoltageSourceAlt1 = 1U }
  *Reference voltage source.*

- enum adc_clock_divider_t {
  kADC_ClockDivider1 = 0U,
  kADC_ClockDivider2 = 1U,
  kADC_ClockDivider4 = 2U,
  kADC_ClockDivider8 = 3U }
  *Clock divider for the converter.*
- enum adc_resolution_mode_t {
  kADC_Resolution8BitMode = 0U,
  kADC_Resolution10BitMode = 1U,
  kADC_Resolution12BitMode = 2U }
  *ADC converter resolution mode.*
- enum adc_clock_source_t {
  kADC_ClockSourceAlt0 = 0U,
  kADC_ClockSourceAlt1 = 1U,
  kADC_ClockSourceAlt2 = 2U,
  kADC_ClockSourceAlt3 = 3U }
  *ADC input Clock source.*
- enum adc_compare_mode_t {
  kADC_CompareDisableMode = 0U,
  kADC_CompareLessMode = 2U,
  kADC_CompareGreaterOrEqualMode = 3U }
  *Compare function mode.*
- enum _adc_status_flags {
  kADC_ActiveFlag = ADC_SC2_ADACT_MASK,
  kADC_FifoEmptyFlag = ADC_SC2_FEMPTY_MASK,
  kADC_FifoFullFlag = ADC_SC2_FFULL_MASK }
  *ADC status flags mask.*
- enum adc_hardware_trigger_mask_mode_t {
  kADC_HWTriggerMaskDisableMode,
  kADC_HWTriggerMaskAutoMode = 1U,
  kADC_HWTriggerMaskEnableMode }
  *Hardware tigger mask mode.*

## Initialization

- void ADC_Init (ADC_Type ∗base, const adc_config_t ∗config)
  *Initializes the ADC module.*
- void ADC_Deinit (ADC_Type ∗base)
  *De-initialize the ADC module.*
- void ADC_GetDefaultConfig (adc_config_t ∗config)
  *Gets an available pre-defined settings for the converter's configuration.*
- static void ADC_EnableHardwareTrigger (ADC_Type ∗base, bool enable)
  *Enable the hardware trigger mode.*
- void ADC_SetHardwareCompare (ADC_Type ∗base, const adc_hardware_compare_config_t ∗config)
  *Configure the hardware compare mode.*
- void ADC_SetFifoConfig (ADC_Type ∗base, const adc_fifo_config_t ∗config)
  *Configure the Fifo mode.*

- void ADC_GetDefaultFIFOConfig (adc_fifo_config_t ∗config)
  *Gets an available pre-defined settings for the FIFO's configuration.*
- void ADC_SetChannelConfig (ADC_Type ∗base, const adc_channel_config_t ∗config)
  *Configures the conversion channel.*
- bool ADC_GetChannelStatusFlags (ADC_Type ∗base)
  *Get the status flags of channel.*
- uint32_t ADC_GetStatusFlags (ADC_Type ∗base)
  *Get the ADC status flags.*
- static void ADC_EnableAnalogInput (ADC_Type ∗base, uint32_t mask, bool enable)
  *Disables the I/O port control of the pins used as analog inputs.*
- static uint32_t ADC_GetChannelConversionValue (ADC_Type ∗base)
  *Gets the conversion value.*
- static void **ADC_SetHardwareTriggerMaskMode** (ADC_Type ∗base, adc_hardware_trigger_mask_mode_t mode)

## 7.3 Data Structure Documentation

### 7.3.1 struct adc_config_t

#### Data Fields

- adc_reference_voltage_source_t referenceVoltageSource
  *Selects the voltage reference source used for conversions.*
- bool enableLowPower
  *Enable low power mode.*
- bool enableLongSampleTime
  *Enable long sample time mode.*
- adc_clock_divider_t clockDivider
  *Select the divider of input clock source.*
- adc_resolution_mode_t ResolutionMode
  *Select the sample resolution mode.*
- adc_clock_source_t clockSource
  *Select the input Clock source.*

#### Field Documentation

##### (1) adc_reference_voltage_source_t adc_config_t::referenceVoltageSource

>

##### (2) bool adc_config_t::enableLowPower

The power is reduced at the expense of maximum clock speed. >

##### (3) bool adc_config_t::enableLongSampleTime

>

#### (4) adc_clock_divider_t adc_config_t::clockDivider

>

#### (5) adc_resolution_mode_t adc_config_t::ResolutionMode

>

#### (6) adc_clock_source_t adc_config_t::clockSource

>

## 7.3.2 struct adc_hardware_compare_config_t

### Data Fields

- uint32_t compareValue
  *Setting the compare value.*
- adc_compare_mode_t compareMode
  *Setting the compare mode.*

#### Field Documentation

#### (1) uint32_t adc_hardware_compare_config_t::compareValue

The value are compared to the conversion result. >

#### (2) adc_compare_mode_t adc_hardware_compare_config_t::compareMode

Refer to "adc_compare_mode_t". >

## 7.3.3 struct adc_fifo_config_t

### Data Fields

- bool enableHWTriggerMultConv
  *The field is valid when FIFO is enabled.Enable hardware trigger multiple conversion.*
- bool enableFifoScanMode
  *The field is valid when FIFO is enabled.*
- bool enableCompareAndMode
  *The field is valid when FIFO is enabled.*
- uint32_t FifoDepth
  *Setting the depth of FIFO.*

#### Field Documentation

#### (1) bool adc_fifo_config_t::enableHWTriggerMultConv

One hardware trigger pulse triggers multiple conversions in fifo mode. >

#### (2) bool adc_fifo_config_t::enableFifoScanMode

Enable the FIFO scan mode. If enable, ADC will repeat using the first FIFO channel as the conversion channel until the result FIFO is fulfilled. >

#### (3) bool adc_fifo_config_t::enableCompareAndMode

If enable, ADC will AND all of compare triggers and set COCO after all of compare triggers occur. If disable, ADC will OR all of compare triggers and set COCO after at least one of compare trigger occurs. >

#### (4) uint32_t adc_fifo_config_t::FifoDepth

Depth of fifo is FifoDepth + 1. When FifoDepth = 0U, the FIFO is DISABLED. When FifoDepth is set to nonzero, the FIFO function is ENABLED and the depth is indicated by the FifoDepth field. >

## 7.3.4 struct adc_channel_config_t

### Data Fields

- uint32_t channelNumber
  *Setting the conversion channel number.*
- bool enableContinuousConversion
  *enables continuous conversions.*
- bool enableInterruptOnConversionCompleted
  *Generate an interrupt request once the conversion is completed.*

### Field Documentation

#### (1) uint32_t adc_channel_config_t::channelNumber

The available range is 0-31. See channel connection information for each chip in Reference Manual document.

#### (2) bool adc_channel_config_t::enableContinuousConversion

>

#### (3) bool adc_channel_config_t::enableInterruptOnConversionCompleted

# 7.4 Macro Definition Documentation

### 7.4.1 #define FSL_ADC_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

Version 2.1.0.

## 7.5 Enumeration Type Documentation

### 7.5.1 enum adc_reference_voltage_source_t

Enumerator

***kADC_ReferenceVoltageSourceAlt0*** Default voltage reference pin pair (VREFH/VREFL). >
***kADC_ReferenceVoltageSourceAlt1*** Analog supply pin pair (VDDA/VSSA). >

### 7.5.2 enum adc_clock_divider_t

Enumerator

***kADC_ClockDivider1*** Divide ration = 1, and clock rate = Input clock. >
***kADC_ClockDivider2*** Divide ration = 2, and clock rate = Input clock / 2. >
***kADC_ClockDivider4*** Divide ration = 3, and clock rate = Input clock / 4. >
***kADC_ClockDivider8*** Divide ration = 4, and clock rate = Input clock / 8. >

### 7.5.3 enum adc_resolution_mode_t

Enumerator

***kADC_Resolution8BitMode*** 8-bit conversion (N = 8). >
***kADC_Resolution10BitMode*** 10-bit conversion (N = 10) >
***kADC_Resolution12BitMode*** 12-bit conversion (N = 12) >

### 7.5.4 enum adc_clock_source_t

Enumerator

***kADC_ClockSourceAlt0*** Bus clock. >
***kADC_ClockSourceAlt1*** Bus clock divided by 2. >
***kADC_ClockSourceAlt2*** Alternate clock (ALTCLK). >
***kADC_ClockSourceAlt3*** Asynchronous clock (ADACK). >

### 7.5.5 enum adc_compare_mode_t

Enumerator

***kADC_CompareDisableMode*** Compare function disabled. >
***kADC_CompareLessMode*** Compare triggers when input is less than compare level. >
***kADC_CompareGreaterOrEqualMode*** Compare triggers when input is greater than or equal to compare level. >

### 7.5.6 enum _adc_status_flags

Enumerator

***kADC_ActiveFlag*** Indicates that a conversion is in progress. >
***kADC_FifoEmptyFlag*** Indicates that ADC result FIFO have no valid new data. >
***kADC_FifoFullFlag*** Indicates that ADC result FIFO is full. >

### 7.5.7 enum adc_hardware_trigger_mask_mode_t

Enumerator

***kADC_HWTriggerMaskDisableMode*** Hardware trigger mask disable and hardware trigger can trigger ADC conversion. >
***kADC_HWTriggerMaskAutoMode*** Hardware trigger mask automatically when data fifo is not empty. >
***kADC_HWTriggerMaskEnableMode*** Hardware trigger mask enable and hardware trigger cannot trigger ADC conversion. >

## 7.6 Function Documentation

### 7.6.1 void ADC_Init ( ADC_Type ∗ *base*, const adc_config_t ∗ *config* )

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *config* | Pointer to configuration structure. See "adc_config_t". |

### 7.6.2 void ADC_Deinit ( ADC_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |

### 7.6.3 void ADC_GetDefaultConfig ( adc_config_t * *config* )

This function initializes the converter configuration structure with available settings. The default values are as follows.

```
*   config->referenceVoltageSource = kADC_ReferenceVoltageSourceAlt0;
*   config->enableLowPower = false;
*   config->enableLongSampleTime = false;
*   config->clockDivider = kADC_ClockDivider1;
*   config->ResolutionMode = kADC_Resolution8BitMode;
*   config->clockSource = kADC_ClockSourceAlt0;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the configuration structure. |

### 7.6.4 static void ADC_EnableHardwareTrigger ( ADC_Type * *base*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *enable* | Switcher of the hardware trigger feature. "true" means enabled, "false" means not enabled. |

### 7.6.5 void ADC_SetHardwareCompare ( ADC_Type * *base*, const adc_hardware_compare_config_t * *config* )

The compare function can be configured to check for an upper or lower limit. After the input is sampled and converted, the result is added to the complement of the compare value (ADC_CV).

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *config* | Pointer to "adc_hardware_compare_config_t" structure. |

### 7.6.6 void ADC_SetFifoConfig ( ADC_Type * *base*, const adc_fifo_config_t * *config* )

The ADC module supports FIFO operation to minimize the interrupts to CPU in order to reduce CP-U loading in ADC interrupt service routines. This module contains two FIFOs to buffer analog input channels and analog results respectively.

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *config* | Pointer to "adc_fifo_config_t" structure. |

### 7.6.7 void ADC_GetDefaultFIFOConfig ( adc_fifo_config_t * *config* )

Parameters

| | |
|---|---|
| *config* | Pointer to the FIFO configuration structure, please refer to adc_fifo_config_t for details. |

### 7.6.8 void ADC_SetChannelConfig ( ADC_Type * *base*, const adc_channel_config_t * *config* )

This operation triggers the conversion when in software trigger mode. When in hardware trigger mode, this API configures the channel while the external trigger source helps to trigger the conversion.

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *config* | Pointer to "adc_channel_config_t" structure. |

### 7.6.9 bool ADC_GetChannelStatusFlags ( ADC_Type * *base* )

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |

Returns

"True" means conversion has completed and "false" means conversion has not completed.

### 7.6.10 uint32_t ADC_GetStatusFlags ( ADC_Type * *base* )

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |

Returns

Flags' mask if indicated flags are asserted. See "_adc_status_flags".

### 7.6.11 static void ADC_EnableAnalogInput ( ADC_Type * *base*, uint32_t *mask*, bool *enable* ) `[inline]`, `[static]`

When a pin control register bit is set, the following conditions are forced for the associated MCU pin: -The output buffer is forced to its high impedance state. -The input buffer is disabled. A read of the I/O port returns a zero for any pin with its input buffer disabled. -The pullup is disabled.

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |
| *mask* | The mask of the pin associated with channel ADx. Valid range is AD0:0x1U ∼ AD15:0x8000U. For example: If enable AD0, AD1 and AD2 pins, mask should be set to 0x7U. |
| *enable* | The "true" means enabled, "false" means not enabled. |

### 7.6.12 static uint32_t ADC_GetChannelConversionValue ( ADC_Type * *base* ) `[inline]`, `[static]`

Parameters

| | |
|---|---|
| *base* | ADC peripheral base address. |

Returns

Conversion value.

# Chapter 8
# Common Driver

## 8.1 Overview

The MCUXpresso SDK provides a driver for the common module of MCUXpresso SDK devices.

### Macros

- #define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1
  *Macro to use the default weak IRQ handler in drivers.*
- #define MAKE_STATUS(group, code) ((((group)∗100L) + (code)))
  *Construct a status code value from a group and code number.*
- #define MAKE_VERSION(major, minor, bugfix) (((major) ∗ 65536L) + ((minor) ∗ 256L) + (bugfix))
  *Construct the version number for drivers.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U
  *No debug console.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U
  *Debug console based on UART.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U
  *Debug console based on LPUART.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U
  *Debug console based on LPSCI.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U
  *Debug console based on USBCDC.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U
  *Debug console based on FLEXCOMM.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U
  *Debug console based on i.MX UART.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U
  *Debug console based on LPC_VUSART.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U
  *Debug console based on LPC_USART.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U
  *Debug console based on SWO.*
- #define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U
  *Debug console based on QSCI.*
- #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
  *Computes the number of elements in an array.*

### Typedefs

- typedef int32_t status_t
  *Type used for all status and error return values.*

## Enumerations

- enum _status_groups {
  kStatusGroup_Generic = 0,
  kStatusGroup_FLASH = 1,
  kStatusGroup_LPSPI = 4,
  kStatusGroup_FLEXIO_SPI = 5,
  kStatusGroup_DSPI = 6,
  kStatusGroup_FLEXIO_UART = 7,
  kStatusGroup_FLEXIO_I2C = 8,
  kStatusGroup_LPI2C = 9,
  kStatusGroup_UART = 10,
  kStatusGroup_I2C = 11,
  kStatusGroup_LPSCI = 12,
  kStatusGroup_LPUART = 13,
  kStatusGroup_SPI = 14,
  kStatusGroup_XRDC = 15,
  kStatusGroup_SEMA42 = 16,
  kStatusGroup_SDHC = 17,
  kStatusGroup_SDMMC = 18,
  kStatusGroup_SAI = 19,
  kStatusGroup_MCG = 20,
  kStatusGroup_SCG = 21,
  kStatusGroup_SDSPI = 22,
  kStatusGroup_FLEXIO_I2S = 23,
  kStatusGroup_FLEXIO_MCULCD = 24,
  kStatusGroup_FLASHIAP = 25,
  kStatusGroup_FLEXCOMM_I2C = 26,
  kStatusGroup_I2S = 27,
  kStatusGroup_IUART = 28,
  kStatusGroup_CSI = 29,
  kStatusGroup_MIPI_DSI = 30,
  kStatusGroup_SDRAMC = 35,
  kStatusGroup_POWER = 39,
  kStatusGroup_ENET = 40,
  kStatusGroup_PHY = 41,
  kStatusGroup_TRGMUX = 42,
  kStatusGroup_SMARTCARD = 43,
  kStatusGroup_LMEM = 44,
  kStatusGroup_QSPI = 45,
  kStatusGroup_DMA = 50,
  kStatusGroup_EDMA = 51,
  kStatusGroup_DMAMGR = 52,
  kStatusGroup_FLEXCAN = 53,
  kStatusGroup_LTC = 54,
  kStatusGroup_FLEXIO_CAMERA = 55,
  kStatusGroup_LPC_SPI = 56,
  kStatusGroup_LPC_USART = 57,
  kStatusGroup_DMIC = 58,
  kStatusGroup_SDIF = 59,

kStatusGroup_POWER_MANAGER = 159 }
*Status group numbers.*
- enum {
kStatus_Success = MAKE_STATUS(kStatusGroup_Generic, 0),
kStatus_Fail = MAKE_STATUS(kStatusGroup_Generic, 1),
kStatus_ReadOnly = MAKE_STATUS(kStatusGroup_Generic, 2),
kStatus_OutOfRange = MAKE_STATUS(kStatusGroup_Generic, 3),
kStatus_InvalidArgument = MAKE_STATUS(kStatusGroup_Generic, 4),
kStatus_Timeout = MAKE_STATUS(kStatusGroup_Generic, 5),
kStatus_NoTransferInProgress,
kStatus_Busy = MAKE_STATUS(kStatusGroup_Generic, 7),
kStatus_NoData }
*Generic status return codes.*

## Functions

- void ∗ SDK_Malloc (size_t size, size_t alignbytes)
  *Allocate memory with given alignment and aligned size.*
- void SDK_Free (void ∗ptr)
  *Free memory.*
- void SDK_DelayAtLeastUs (uint32_t delayTime_us, uint32_t coreClock_Hz)
  *Delay at least for some time.*

## Driver version

- #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))
  *common driver version.*

## Min/max macros

- #define **MIN**(a, b) (((a) < (b)) ? (a) : (b))
- #define **MAX**(a, b) (((a) > (b)) ? (a) : (b))

## UINT16_MAX/UINT32_MAX value

- #define **UINT16_MAX** ((uint16_t)-1)
- #define **UINT32_MAX** ((uint32_t)-1)

## Suppress fallthrough warning macro

- #define **SUPPRESS_FALL_THROUGH_WARNING**()

## 8.2 Macro Definition Documentation

### 8.2.1 #define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1

### 8.2.2 #define MAKE_STATUS( *group*, *code* ) ((((group)∗100L) + (code)))

### 8.2.3 #define MAKE_VERSION( *major,* *minor,* *bugfix* ) (((major) ∗ 65536L) + ((minor) ∗ 256L) + (bugfix))

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
```

### 8.2.4 #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))

### 8.2.5 #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U

### 8.2.6 #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U

### 8.2.7 #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U

### 8.2.8 #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U

### 8.2.9 #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U

### 8.2.10 #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U

### 8.2.11 #define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U

### 8.2.12 #define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U

### 8.2.13 #define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U

### 8.2.14 #define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U

### 8.2.15 #define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U

### 8.2.16 #define ARRAY_SIZE( *x* ) (sizeof(x) / sizeof((x)[0]))

## 8.3 Typedef Documentation

### 8.3.1 typedef int32_t status_t

## 8.4 Enumeration Type Documentation

### 8.4.1 enum _status_groups

Enumerator

***kStatusGroup_Generic*** Group number for generic status codes.
***kStatusGroup_FLASH*** Group number for FLASH status codes.
***kStatusGroup_LPSPI*** Group number for LPSPI status codes.
***kStatusGroup_FLEXIO_SPI*** Group number for FLEXIO SPI status codes.
***kStatusGroup_DSPI*** Group number for DSPI status codes.
***kStatusGroup_FLEXIO_UART*** Group number for FLEXIO UART status codes.
***kStatusGroup_FLEXIO_I2C*** Group number for FLEXIO I2C status codes.
***kStatusGroup_LPI2C*** Group number for LPI2C status codes.
***kStatusGroup_UART*** Group number for UART status codes.
***kStatusGroup_I2C*** Group number for UART status codes.
***kStatusGroup_LPSCI*** Group number for LPSCI status codes.
***kStatusGroup_LPUART*** Group number for LPUART status codes.
***kStatusGroup_SPI*** Group number for SPI status code.
***kStatusGroup_XRDC*** Group number for XRDC status code.
***kStatusGroup_SEMA42*** Group number for SEMA42 status code.
***kStatusGroup_SDHC*** Group number for SDHC status code.
***kStatusGroup_SDMMC*** Group number for SDMMC status code.
***kStatusGroup_SAI*** Group number for SAI status code.
***kStatusGroup_MCG*** Group number for MCG status codes.
***kStatusGroup_SCG*** Group number for SCG status codes.
***kStatusGroup_SDSPI*** Group number for SDSPI status codes.
***kStatusGroup_FLEXIO_I2S*** Group number for FLEXIO I2S status codes.
***kStatusGroup_FLEXIO_MCULCD*** Group number for FLEXIO LCD status codes.
***kStatusGroup_FLASHIAP*** Group number for FLASHIAP status codes.
***kStatusGroup_FLEXCOMM_I2C*** Group number for FLEXCOMM I2C status codes.
***kStatusGroup_I2S*** Group number for I2S status codes.
***kStatusGroup_IUART*** Group number for IUART status codes.
***kStatusGroup_CSI*** Group number for CSI status codes.
***kStatusGroup_MIPI_DSI*** Group number for MIPI DSI status codes.
***kStatusGroup_SDRAMC*** Group number for SDRAMC status codes.
***kStatusGroup_POWER*** Group number for POWER status codes.
***kStatusGroup_ENET*** Group number for ENET status codes.
***kStatusGroup_PHY*** Group number for PHY status codes.
***kStatusGroup_TRGMUX*** Group number for TRGMUX status codes.
***kStatusGroup_SMARTCARD*** Group number for SMARTCARD status codes.
***kStatusGroup_LMEM*** Group number for LMEM status codes.
***kStatusGroup_QSPI*** Group number for QSPI status codes.
***kStatusGroup_DMA*** Group number for DMA status codes.
***kStatusGroup_EDMA*** Group number for EDMA status codes.
***kStatusGroup_DMAMGR*** Group number for DMAMGR status codes.

***kStatusGroup_FLEXCAN*** Group number for FlexCAN status codes.
***kStatusGroup_LTC*** Group number for LTC status codes.
***kStatusGroup_FLEXIO_CAMERA*** Group number for FLEXIO CAMERA status codes.
***kStatusGroup_LPC_SPI*** Group number for LPC_SPI status codes.
***kStatusGroup_LPC_USART*** Group number for LPC_USART status codes.
***kStatusGroup_DMIC*** Group number for DMIC status codes.
***kStatusGroup_SDIF*** Group number for SDIF status codes.
***kStatusGroup_SPIFI*** Group number for SPIFI status codes.
***kStatusGroup_OTP*** Group number for OTP status codes.
***kStatusGroup_MCAN*** Group number for MCAN status codes.
***kStatusGroup_CAAM*** Group number for CAAM status codes.
***kStatusGroup_ECSPI*** Group number for ECSPI status codes.
***kStatusGroup_USDHC*** Group number for USDHC status codes.
***kStatusGroup_LPC_I2C*** Group number for LPC_I2C status codes.
***kStatusGroup_DCP*** Group number for DCP status codes.
***kStatusGroup_MSCAN*** Group number for MSCAN status codes.
***kStatusGroup_ESAI*** Group number for ESAI status codes.
***kStatusGroup_FLEXSPI*** Group number for FLEXSPI status codes.
***kStatusGroup_MMDC*** Group number for MMDC status codes.
***kStatusGroup_PDM*** Group number for MIC status codes.
***kStatusGroup_SDMA*** Group number for SDMA status codes.
***kStatusGroup_ICS*** Group number for ICS status codes.
***kStatusGroup_SPDIF*** Group number for SPDIF status codes.
***kStatusGroup_LPC_MINISPI*** Group number for LPC_MINISPI status codes.
***kStatusGroup_HASHCRYPT*** Group number for Hashcrypt status codes.
***kStatusGroup_LPC_SPI_SSP*** Group number for LPC_SPI_SSP status codes.
***kStatusGroup_I3C*** Group number for I3C status codes.
***kStatusGroup_LPC_I2C_1*** Group number for LPC_I2C_1 status codes.
***kStatusGroup_NOTIFIER*** Group number for NOTIFIER status codes.
***kStatusGroup_DebugConsole*** Group number for debug console status codes.
***kStatusGroup_SEMC*** Group number for SEMC status codes.
***kStatusGroup_ApplicationRangeStart*** Starting number for application groups.
***kStatusGroup_IAP*** Group number for IAP status codes.
***kStatusGroup_SFA*** Group number for SFA status codes.
***kStatusGroup_SPC*** Group number for SPC status codes.
***kStatusGroup_PUF*** Group number for PUF status codes.
***kStatusGroup_TOUCH_PANEL*** Group number for touch panel status codes.
***kStatusGroup_HAL_GPIO*** Group number for HAL GPIO status codes.
***kStatusGroup_HAL_UART*** Group number for HAL UART status codes.
***kStatusGroup_HAL_TIMER*** Group number for HAL TIMER status codes.
***kStatusGroup_HAL_SPI*** Group number for HAL SPI status codes.
***kStatusGroup_HAL_I2C*** Group number for HAL I2C status codes.
***kStatusGroup_HAL_FLASH*** Group number for HAL FLASH status codes.
***kStatusGroup_HAL_PWM*** Group number for HAL PWM status codes.
***kStatusGroup_HAL_RNG*** Group number for HAL RNG status codes.

***kStatusGroup_HAL_I2S*** Group number for HAL I2S status codes.
***kStatusGroup_TIMERMANAGER*** Group number for TiMER MANAGER status codes.
***kStatusGroup_SERIALMANAGER*** Group number for SERIAL MANAGER status codes.
***kStatusGroup_LED*** Group number for LED status codes.
***kStatusGroup_BUTTON*** Group number for BUTTON status codes.
***kStatusGroup_EXTERN_EEPROM*** Group number for EXTERN EEPROM status codes.
***kStatusGroup_SHELL*** Group number for SHELL status codes.
***kStatusGroup_MEM_MANAGER*** Group number for MEM MANAGER status codes.
***kStatusGroup_LIST*** Group number for List status codes.
***kStatusGroup_OSA*** Group number for OSA status codes.
***kStatusGroup_COMMON_TASK*** Group number for Common task status codes.
***kStatusGroup_MSG*** Group number for messaging status codes.
***kStatusGroup_SDK_OCOTP*** Group number for OCOTP status codes.
***kStatusGroup_SDK_FLEXSPINOR*** Group number for FLEXSPINOR status codes.
***kStatusGroup_CODEC*** Group number for codec status codes.
***kStatusGroup_ASRC*** Group number for codec status ASRC.
***kStatusGroup_OTFAD*** Group number for codec status codes.
***kStatusGroup_SDIOSLV*** Group number for SDIOSLV status codes.
***kStatusGroup_MECC*** Group number for MECC status codes.
***kStatusGroup_ENET_QOS*** Group number for ENET_QOS status codes.
***kStatusGroup_LOG*** Group number for LOG status codes.
***kStatusGroup_I3CBUS*** Group number for I3CBUS status codes.
***kStatusGroup_QSCI*** Group number for QSCI status codes.
***kStatusGroup_SNT*** Group number for SNT status codes.
***kStatusGroup_QUEUEDSPI*** Group number for QSPI status codes.
***kStatusGroup_POWER_MANAGER*** Group number for POWER_MANAGER status codes.

### 8.4.2 anonymous enum

Enumerator

***kStatus_Success*** Generic status for Success.
***kStatus_Fail*** Generic status for Fail.
***kStatus_ReadOnly*** Generic status for read only failure.
***kStatus_OutOfRange*** Generic status for out of range access.
***kStatus_InvalidArgument*** Generic status for invalid argument check.
***kStatus_Timeout*** Generic status for timeout.
***kStatus_NoTransferInProgress*** Generic status for no transfer in progress.
***kStatus_Busy*** Generic status for module is busy.
***kStatus_NoData*** Generic status for no data is found for the operation.

## 8.5 Function Documentation

### 8.5.1 void∗ SDK_Malloc ( size_t *size,* size_t *alignbytes* )

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. |

### 8.5.2 void SDK_Free ( void * *ptr* )

Parameters

| | |
|---|---|
| *ptr* | The memory to be release. |

### 8.5.3 void SDK_DelayAtLeastUs ( uint32_t *delayTime_us,* uint32_t *coreClock_Hz* )

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. |

# Chapter 9
# FTMRx Flash Driver

## 9.1 Overview

The flash provides the FTMRx Flash driver of MCUXpresso SDK devices with the FTMRx Flash module inside. The flash driver provides general APIs to handle specific operations on the FTMRx Flash module. The user can use those APIs directly in the application. In addition, it provides internal functions called by the driver. Although these functions are not meant to be called from the user's application directly, the APIs can still be used.

### Data Structures

- struct pflash_protection_status_t
  *PFlash protection status - full. More...*
- struct flash_prefetch_speculation_status_t
  *Flash prefetch speculation status. More...*
- struct flash_protection_config_t
  *Active flash protection information for the current operation. More...*
- struct flash_operation_config_t
  *Active flash information for the current operation. More...*
- union function_run_command_t
  *Flash execute-in-RAM command. More...*
- struct flash_execute_in_ram_function_config_t
  *Flash execute-in-RAM function information. More...*
- struct flash_config_t
  *Flash driver state information. More...*

### Typedefs

- typedef void(∗ flash_callback_t )(void)
  *A callback type used for the Pflash block.*

### Enumerations

- enum flash_user_margin_value_t {
  kFLASH_ReadMarginValueNormal = 0x0000U,
  kFLASH_UserMarginValue1 = 0x0001U,
  kFLASH_UserMarginValue0 = 0x0002U }
  *Enumeration for supported flash user margin levels.*
- enum flash_factory_margin_value_t {
  kFLASH_FactoryMarginValue1 = 0x0003U,
  kFLASH_FactoryMarginValue0 = 0x0004U }
  *Enumeration for supported factory margin levels.*

- enum flash_margin_value_t {
  kFLASH_MarginValueNormal,
  kFLASH_MarginValueUser,
  kFLASH_MarginValueFactory,
  kFLASH_MarginValueInvalid }
  *Enumeration for supported flash margin levels.*
- enum flash_security_state_t {
  kFLASH_SecurityStateNotSecure,
  kFLASH_SecurityStateBackdoorEnabled,
  kFLASH_SecurityStateBackdoorDisabled }
  *Enumeration for the three possible flash security states.*
- enum flash_protection_state_t {
  kFLASH_ProtectionStateUnprotected,
  kFLASH_ProtectionStateProtected,
  kFLASH_ProtectionStateMixed }
  *Enumeration for the three possible flash protection levels.*
- enum flash_property_tag_t {
  kFLASH_PropertyPflashSectorSize = 0x00U,
  kFLASH_PropertyPflashTotalSize = 0x01U,
  kFLASH_PropertyPflashBlockSize = 0x02U,
  kFLASH_PropertyPflashBlockCount = 0x03U,
  kFLASH_PropertyPflashBlockBaseAddr = 0x04U,
  kFLASH_PropertyPflashFacSupport = 0x05U,
  kFLASH_PropertyEepromTotalSize = 0x15U,
  kFLASH_PropertyFlashMemoryIndex = 0x20U,
  kFLASH_PropertyFlashCacheControllerIndex = 0x21U,
  kFLASH_PropertyEepromBlockBaseAddr = 0x22U,
  kFLASH_PropertyEepromSectorSize = 0x23U,
  kFLASH_PropertyEepromBlockSize = 0x24U,
  kFLASH_PropertyEepromBlockCount = 0x25U,
  kFLASH_PropertyFlashClockFrequency = 0x26U }
  *Enumeration for various flash properties.*
- enum {
  kFLASH_ExecuteInRamFunctionMaxSizeInWords = 16U,
  kFLASH_ExecuteInRamFunctionTotalNum = 2U }
  *Constants for execute-in-RAM flash function.*
- enum flash_memory_index_t {
  kFLASH_MemoryIndexPrimaryFlash = 0x00U,
  kFLASH_MemoryIndexSecondaryFlash = 0x01U }
  *Enumeration for the flash memory index.*
- enum flash_cache_controller_index_t {
  kFLASH_CacheControllerIndexForCore0 = 0x00U,
  kFLASH_CacheControllerIndexForCore1 = 0x01U }
  *Enumeration for the flash cache controller index.*
- enum flash_prefetch_speculation_option_t
  *Enumeration for the two possible options of flash prefetch speculation.*
- enum flash_cache_clear_process_t {

kFLASH_CacheClearProcessPre = 0x00U,
kFLASH_CacheClearProcessPost = 0x01U }
*Flash cache clear process code.*

## Flash version

- enum _flash_driver_version_constants {
kFLASH_DriverVersionName = ’F’,
kFLASH_DriverVersionMajor = 2,
kFLASH_DriverVersionMinor = 1,
kFLASH_DriverVersionBugfix = 1 }
*Flash driver version for ROM.*
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
*Constructs the version number for drivers.*
- #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
*Flash driver version for SDK.*

## Flash configuration

- #define FLASH_SSD_CONFIG_ENABLE_EEPROM_SUPPORT 0
*Indicates whether to support EEPROM in the Flash driver.*
- #define FLASH_SSD_IS_EEPROM_ENABLED FLASH_SSD_CONFIG_ENABLE_EEPROM_SUPPORT
*Indicates whether the EEPROM is enabled in the Flash driver.*
- #define FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT 1
*Indicates whether to support Secondary flash in the Flash driver.*
- #define FLASH_SSD_IS_SECONDARY_FLASH_ENABLED (0)
*Indicates whether the secondary flash is supported in the Flash driver.*
- #define FLASH_DRIVER_IS_FLASH_RESIDENT 1
*Flash driver location.*
- #define FLASH_DRIVER_IS_EXPORTED 0
*Flash Driver Export option.*
- #define FLASH_ENABLE_STALLING_FLASH_CONTROLLER 1
*Enable flash stalling controller.*

## Flash status

- enum {
kStatus_FLASH_Success = MAKE_STATUS(kStatusGroupGeneric, 0),
kStatus_FLASH_InvalidArgument = MAKE_STATUS(kStatusGroupGeneric, 4),
kStatus_FLASH_SizeError = MAKE_STATUS(kStatusGroupFlashDriver, 0),
kStatus_FLASH_AlignmentError,
kStatus_FLASH_AddressError = MAKE_STATUS(kStatusGroupFlashDriver, 2),
kStatus_FLASH_AccessError,
kStatus_FLASH_ProtectionViolation,
kStatus_FLASH_CommandFailure,
kStatus_FLASH_UnknownProperty = MAKE_STATUS(kStatusGroupFlashDriver, 6),
kStatus_FLASH_EraseKeyError = MAKE_STATUS(kStatusGroupFlashDriver, 7),
kStatus_FLASH_RegionExecuteOnly,
kStatus_FLASH_ExecuteInRamFunctionNotReady,
kStatus_FLASH_PartitionStatusUpdateFailure,
kStatus_FLASH_SetFlexramAsEepromError,
kStatus_FLASH_RecoverFlexramAsRamError,
kStatus_FLASH_SetFlexramAsRamError = MAKE_STATUS(kStatusGroupFlashDriver, 13),
kStatus_FLASH_RecoverFlexramAsEepromError,
kStatus_FLASH_CommandNotSupported = MAKE_STATUS(kStatusGroupFlashDriver, 15),
kStatus_FLASH_SwapSystemNotInUninitialized,
kStatus_FLASH_SwapIndicatorAddressError,
kStatus_FLASH_ReadOnlyProperty = MAKE_STATUS(kStatusGroupFlashDriver, 18),
kStatus_FLASH_InvalidPropertyValue,
kStatus_FLASH_InvalidSpeculationOption,
kStatus_FLASH_ClockDivider = MAKE_STATUS(kStatusGroupFlashDriver, 21),
kStatus_FLASH_EepromDoubleBitFault,
kStatus_FLASH_EepromSingleBitFault }
  *Flash driver status codes.*
- #define kStatusGroupGeneric 0
  *Flash driver status group.*
- #define **kStatusGroupFlashDriver** 1
- #define MAKE_STATUS(group, code) ((((group)∗100) + (code)))
  *Constructs a status code value from a group and a code number.*

## Flash API key

- enum _flash_driver_api_keys { kFLASH_ApiEraseKey = FOUR_CHAR_CODE('k', 'f', 'e', 'k') }
  *Enumeration for Flash driver API keys.*
- #define FOUR_CHAR_CODE(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))
  *Constructs the four character code for the Flash driver API key.*

## Initialization

- status_t FLASH_Init (flash_config_t ∗config)
  *Initializes the global flash properties structure members.*
- status_t FLASH_SetCallback (flash_config_t ∗config, flash_callback_t callback)

*Sets the desired flash callback function.*
- status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t ∗config)
  *Prepares flash execute-in-RAM functions.*

## Erasing

- status_t FLASH_EraseAll (flash_config_t ∗config, uint32_t key)
  *Erases entire flash.*
- 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.*
- status_t FLASH_EraseAllUnsecure (flash_config_t ∗config, uint32_t key)
  *Erases the entire flash, including protected sectors.*

## Programming

- status_t FLASH_Program (flash_config_t ∗config, uint32_t start, uint32_t ∗src, uint32_t lengthInBytes)
  *Programs flash with data at locations passed in through parameters.*
- status_t FLASH_ProgramOnce (flash_config_t ∗config, uint32_t index, uint32_t ∗src, uint32_t lengthInBytes)
  *Programs Program Once Field through parameters.*

## Reading

- status_t FLASH_ReadOnce (flash_config_t ∗config, uint32_t index, uint32_t ∗dst, uint32_t lengthInBytes)
  *Reads the Program Once Field through parameters.*

## Security

- status_t FLASH_GetSecurityState (flash_config_t ∗config, flash_security_state_t ∗state)
  *Returns the security state via the pointer passed into the function.*
- status_t FLASH_SecurityBypass (flash_config_t ∗config, const uint8_t ∗backdoorKey)
  *Allows users to bypass security with a backdoor key.*

## Verification

- status_t FLASH_VerifyEraseAll (flash_config_t ∗config, flash_margin_value_t margin)
  *Verifies erasure of the entire flash at a specified margin level.*
- status_t FLASH_VerifyErase (flash_config_t ∗config, uint32_t start, uint32_t lengthInBytes, flash_margin_value_t margin)
  *Verifies an erasure of the desired flash area at a specified margin level.*

## Protection

- status_t FLASH_IsProtected (flash_config_t ∗config, uint32_t start, uint32_t lengthInBytes, flash_protection_state_t ∗protection_state)
  *Returns the protection state of the desired flash area via the pointer passed into the function.*

## Properties

- status_t FLASH_GetProperty (flash_config_t *config, flash_property_tag_t whichProperty, uint32_t *value)
  *Returns the desired flash property.*
- status_t FLASH_SetProperty (flash_config_t *config, flash_property_tag_t whichProperty, uint32_t value)
  *Sets the desired flash property.*

## Flash Protection Utilities

- status_t FLASH_PflashSetProtection (flash_config_t *config, pflash_protection_status_t *protectStatus)
  *Sets the PFlash Protection to the intended protection status.*
- status_t FLASH_PflashGetProtection (flash_config_t *config, pflash_protection_status_t *protectStatus)
  *Gets the PFlash protection status.*

## Flash Speculation Utilities

- status_t FLASH_PflashSetPrefetchSpeculation (flash_prefetch_speculation_status_t *speculationStatus)
  *Sets the PFlash prefetch speculation to the intended speculation status.*
- status_t FLASH_PflashGetPrefetchSpeculation (flash_prefetch_speculation_status_t *speculationStatus)
  *Gets the PFlash prefetch speculation status.*

# 9.2 Data Structure Documentation

## 9.2.1 struct pflash_protection_status_t

### Data Fields

- uint8_t fprotvalue
  *FPROT[7:0] .*

#### Field Documentation

##### (1) uint8_t pflash_protection_status_t::fprotvalue

## 9.2.2 struct flash_prefetch_speculation_status_t

### Data Fields

- flash_prefetch_speculation_option_t instructionOption
  *Instruction speculation.*
- flash_prefetch_speculation_option_t dataOption
  *Data speculation.*

#### Field Documentation

##### (1) flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::instructionOption

##### (2) flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::dataOption

## 9.2.3 struct flash_protection_config_t

### Data Fields

- uint32_t lowRegionStart
  *Start address of flash protection low region.*
- uint32_t lowRegionEnd
  *End address of flash protection low region.*
- uint32_t highRegionStart
  *Start address of flash protection high region.*
- uint32_t highRegionEnd
  *End address of flash protection high region.*

#### Field Documentation

##### (1) uint32_t flash_protection_config_t::lowRegionStart

##### (2) uint32_t flash_protection_config_t::lowRegionEnd

##### (3) uint32_t flash_protection_config_t::highRegionStart

##### (4) uint32_t flash_protection_config_t::highRegionEnd

## 9.2.4 struct flash_operation_config_t

### Data Fields

- uint32_t convertedAddress
  *A converted address for the current flash type.*
- uint32_t activeSectorSize
  *A sector size of the current flash type.*
- uint32_t activeBlockSize
  *A block size of the current flash type.*
- uint32_t blockWriteUnitSize
  *The write unit size.*
- uint32_t sectorCmdAddressAligment
  *An erase sector command address alignment.*
- uint32_t sectionCmdAddressAligment
  *A program/verify section command address alignment.*
- uint32_t programCmdAddressAligment
  *A program flash command address alignment.*

#### Field Documentation

#### (1) uint32_t flash_operation_config_t::convertedAddress

#### (2) uint32_t flash_operation_config_t::activeSectorSize

#### (3) uint32_t flash_operation_config_t::activeBlockSize

#### (4) uint32_t flash_operation_config_t::blockWriteUnitSize

#### (5) uint32_t flash_operation_config_t::sectorCmdAddressAligment

#### (6) uint32_t flash_operation_config_t::sectionCmdAddressAligment

#### (7) uint32_t flash_operation_config_t::programCmdAddressAligment

## 9.2.5 union function_run_command_t

## 9.2.6 struct flash_execute_in_ram_function_config_t

### Data Fields

- uint32_t activeFunctionCount
  *Number of available execute-in-RAM functions.*
- function_run_command_t runCmdFuncAddr
  *Execute-in-RAM function: flash_run_command.*

#### Field Documentation

#### (1) uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

#### (2) function_run_command_t flash_execute_in_ram_function_config_t::runCmdFuncAddr

## 9.2.7 struct flash_config_t

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

### Data Fields

- uint32_t PFlashBlockBase
  *A base address of the first PFlash block.*
- uint32_t PFlashTotalSize
  *The size of the combined PFlash block.*
- uint8_t PFlashBlockCount
  *A number of PFlash blocks.*
- uint8_t FlashMemoryIndex
  *0 - primary flash; 1 - secondary flash*
- uint8_t FlashCacheControllerIndex

*0 - Controller for core 0; 1 - Controller for core 1*

- uint8_t Reserved0
  *Reserved field 0.*
- uint32_t PFlashSectorSize
  *The size in bytes of a sector of PFlash.*
- flash_callback_t PFlashCallback
  *The callback function for the flash API.*
- uint32_t * flashExecuteInRamFunctionInfo
  *An information structure of the flash execute-in-RAM function.*
- uint32_t EEpromTotalSize
  *For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM.*
- uint32_t EEpromBlockBase
  *This is the base address of the Eeprom.*
- uint8_t EEpromBlockCount
  *A number of EEPROM blocks.*
- uint8_t EEpromSectorSize
  *The size in bytes of a sector of EEPROM.*
- uint8_t Reserved1 [2]
  *Reserved field 1.*
- uint32_t PFlashClockFreq
  *The flash peripheral clock frequency.*
- uint32_t PFlashMarginLevel
  *The margin level.*

### Field Documentation

#### (1) uint32_t flash_config_t::PFlashTotalSize

#### (2) uint8_t flash_config_t::PFlashBlockCount

#### (3) uint32_t flash_config_t::PFlashSectorSize

#### (4) flash_callback_t flash_config_t::PFlashCallback

#### (5) uint32_t* flash_config_t::flashExecuteInRamFunctionInfo

#### (6) uint32_t flash_config_t::EEpromTotalSize

For the non-FlexNVM device, this field is unused

#### (7) uint32_t flash_config_t::EEpromBlockBase

For the non-Eeprom device, this field is unused

#### (8) uint8_t flash_config_t::EEpromBlockCount

For the non-Eeprom device, this field is unused

**(9) uint8_t flash_config_t::EEpromSectorSize**

For the non-Eeprom device, this field is unused

## 9.3 Macro Definition Documentation

### 9.3.1 #define MAKE_VERSION( *major*, *minor*, *bugfix* ) (((major) << 16) | ((minor) << 8) | (bugfix))

### 9.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

Version 2.1.2.

### 9.3.3 #define FLASH_SSD_CONFIG_ENABLE_EEPROM_SUPPORT 0

Disables the EEPROM support.

### 9.3.4 #define FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT 1

Enables the secondary flash support by default.

### 9.3.5 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

### 9.3.6 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the MCUXpresso SDK application.

### 9.3.7 #define kStatusGroupGeneric 0

### 9.3.8 #define MAKE_STATUS( *group*, *code* ) ((((group)*100) + (code)))

### 9.3.9 #define FOUR_CHAR_CODE( *a*, *b*, *c*, *d* ) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))

## 9.4 Enumeration Type Documentation

### 9.4.1 enum _flash_driver_version_constants

Enumerator

***kFLASH_DriverVersionName*** Flash driver version name.
***kFLASH_DriverVersionMajor*** Major flash driver version.
***kFLASH_DriverVersionMinor*** Minor flash driver version.
***kFLASH_DriverVersionBugfix*** Bugfix for flash driver version.

### 9.4.2 anonymous enum

Enumerator

***kStatus_FLASH_Success*** API is executed successfully.
***kStatus_FLASH_InvalidArgument*** Invalid argument.
***kStatus_FLASH_SizeError*** Error size.
***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 bound addresses.
***kStatus_FLASH_ProtectionViolation*** The program/erase operation is requested to execute on protected areas.
***kStatus_FLASH_CommandFailure*** Run-time error during command execution.
***kStatus_FLASH_UnknownProperty*** Unknown property.
***kStatus_FLASH_EraseKeyError*** API erase key is invalid.
***kStatus_FLASH_RegionExecuteOnly*** The current region is execute-only.
***kStatus_FLASH_ExecuteInRamFunctionNotReady*** Execute-in-RAM function is not available.
***kStatus_FLASH_PartitionStatusUpdateFailure*** Failed to update partition status.
***kStatus_FLASH_SetFlexramAsEepromError*** Failed to set FlexRAM as EEPROM.
***kStatus_FLASH_RecoverFlexramAsRamError*** Failed to recover FlexRAM as RAM.
***kStatus_FLASH_SetFlexramAsRamError*** Failed to set FlexRAM as RAM.
***kStatus_FLASH_RecoverFlexramAsEepromError*** Failed to recover FlexRAM as EEPROM.
***kStatus_FLASH_CommandNotSupported*** Flash API is not supported.
***kStatus_FLASH_SwapSystemNotInUninitialized*** Swap system is not in an uninitialzed state.
***kStatus_FLASH_SwapIndicatorAddressError*** The swap indicator address is invalid.
***kStatus_FLASH_ReadOnlyProperty*** The flash property is read-only.
***kStatus_FLASH_InvalidPropertyValue*** The flash property value is out of range.
***kStatus_FLASH_InvalidSpeculationOption*** The option of flash prefetch speculation is invalid.
***kStatus_FLASH_ClockDivider*** Flash clock prescaler is wrong.
***kStatus_FLASH_EepromDoubleBitFault*** A double bit fault was detected in the stored parity.
***kStatus_FLASH_EepromSingleBitFault*** A single bit fault was detected in the stored parity.

### 9.4.3 enum _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.

Enumerator

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

### 9.4.4 enum flash_user_margin_value_t

Enumerator

***kFLASH_ReadMarginValueNormal*** Use the 'normal' read level for 1s.
***kFLASH_UserMarginValue1*** Apply the 'User' margin to the normal read-1 level.
***kFLASH_UserMarginValue0*** Apply the 'User' margin to the normal read-0 level.

### 9.4.5 enum flash_factory_margin_value_t

Enumerator

***kFLASH_FactoryMarginValue1*** Apply the 'Factory' margin to the normal read-1 level.
***kFLASH_FactoryMarginValue0*** Apply the 'Factory' margin to the normal read-0 level.

### 9.4.6 enum flash_margin_value_t

Enumerator

***kFLASH_MarginValueNormal*** Use the 'normal' read level for 1s.
***kFLASH_MarginValueUser*** Apply the 'User' margin to the normal read-1 level.
***kFLASH_MarginValueFactory*** Apply the 'Factory' margin to the normal read-1 level.
***kFLASH_MarginValueInvalid*** Not real margin level, Used to determine the range of valid margin level.

### 9.4.7 enum flash_security_state_t

Enumerator

***kFLASH_SecurityStateNotSecure*** Flash is not secure.
***kFLASH_SecurityStateBackdoorEnabled*** Flash backdoor is enabled.
***kFLASH_SecurityStateBackdoorDisabled*** Flash backdoor is disabled.

### 9.4.8 enum flash_protection_state_t

Enumerator

***kFLASH_ProtectionStateUnprotected*** Flash region is not protected.
***kFLASH_ProtectionStateProtected*** Flash region is protected.
***kFLASH_ProtectionStateMixed*** Flash is mixed with protected and unprotected region.

### 9.4.9 enum flash_property_tag_t

Enumerator

***kFLASH_PropertyPflashSectorSize*** Pflash sector size property.
***kFLASH_PropertyPflashTotalSize*** Pflash total size property.
***kFLASH_PropertyPflashBlockSize*** Pflash block size property.
***kFLASH_PropertyPflashBlockCount*** Pflash block count property.
***kFLASH_PropertyPflashBlockBaseAddr*** Pflash block base address property.
***kFLASH_PropertyPflashFacSupport*** Pflash fac support property.
***kFLASH_PropertyEepromTotalSize*** EEPROM total size property.
***kFLASH_PropertyFlashMemoryIndex*** Flash memory index property.
***kFLASH_PropertyFlashCacheControllerIndex*** Flash cache controller index property.
***kFLASH_PropertyEepromBlockBaseAddr*** EEPROM block base address property.
***kFLASH_PropertyEepromSectorSize*** EEPROM sector size property.
***kFLASH_PropertyEepromBlockSize*** EEPROM block size property.
***kFLASH_PropertyEepromBlockCount*** EEPROM block count property.
***kFLASH_PropertyFlashClockFrequency*** Flash peripheral clock property.

### 9.4.10 anonymous enum

_flash_execute_in_ram_function_constants

Enumerator

***kFLASH_ExecuteInRamFunctionMaxSizeInWords*** The maximum size of execute-in-RAM function.
***kFLASH_ExecuteInRamFunctionTotalNum*** Total number of execute-in-RAM functions.

### 9.4.11 enum flash_memory_index_t

Enumerator

***kFLASH_MemoryIndexPrimaryFlash*** Current flash memory is primary flash.
***kFLASH_MemoryIndexSecondaryFlash*** Current flash memory is secondary flash.

### 9.4.12 enum flash_cache_controller_index_t

Enumerator

***kFLASH_CacheControllerIndexForCore0*** Current flash cache controller is for core 0.
***kFLASH_CacheControllerIndexForCore1*** Current flash cache controller is for core 1.

### 9.4.13 enum flash_cache_clear_process_t

Enumerator

***kFLASH_CacheClearProcessPre*** Pre flash cache clear process.
***kFLASH_CacheClearProcessPost*** Post flash cache clear process.

## 9.5 Function Documentation

### 9.5.1 status_t FLASH_Init ( flash_config_t * *config* )

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_ClockDivider* | Flash clock prescaler is wrong. |
| *kStatus_FLASH_ExecuteInRamFunctionNotReady* | Execute-in-RAM function is not available. |

### 9.5.2 status_t FLASH_SetCallback ( flash_config_t * *config,* flash_callback_t *callback* )

Parameters

| | |
|---|---|
| *config* | Pointer to the storage for the driver runtime state. |
| *callback* | A callback function to be stored in the driver. |

Return values

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

### 9.5.3 status_t FLASH_PrepareExecuteInRamFunctions ( flash_config_t ∗ *config* )

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. |

### 9.5.4 status_t FLASH_EraseAll ( flash_config_t ∗ *config,* uint32_t *key* )

Parameters

| | |
|---|---|
| *config* | Pointer to the storage for the driver runtime state. |
| *key* | A value used to validate all flash erase APIs. |

Return values

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

| | |
|---|---|
| *kStatus_FLASH_Erase-KeyError* | API erase key is invalid. |
| *kStatus_FLASH_Execute-InRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_Access-Error* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_-ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during command execution. |
| *kStatus_FLASH_Eeprom-SingleBitFault* | EEPROM single bit fault error code. |
| *kStatus_FLASH_Eeprom-DoubleBitFault* | EEPROM double bit fault error code. |

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

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 does not need to be sector-aligned but must be word-aligned. |
| *lengthInBytes* | The length, given in bytes (not words or long-words) to be erased. Must be word-aligned. |
| *key* | The value used to validate all flash erase APIs. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |

| | |
|---|---|
| kStatus_FLASH_-AlignmentError | The parameter is not aligned with the specified baseline. |
| kStatus_FLASH_Address-Error | The address is out of range. |
| kStatus_FLASH_Erase-KeyError | The API erase key is invalid. |
| kStatus_FLASH_Execute-InRamFunctionNotReady | Execute-in-RAM function is not available. |
| kStatus_FLASH_Access-Error | Invalid instruction codes and out-of bounds addresses. |
| kStatus_FLASH_-ProtectionViolation | The program/erase operation is requested to execute on protected areas. |
| kStatus_FLASH_-CommandFailure | Run-time error during the command execution. |

### 9.5.6 status_t FLASH_EraseAllUnsecure ( flash_config_t * *config*, uint32_t *key* )

Parameters

| | |
|---|---|
| *config* | Pointer to the storage for the driver runtime state. |
| *key* | A value used to validate all flash erase APIs. |

Return values

| | |
|---|---|
| kStatus_FLASH_Success | API was executed successfully. |
| kStatus_FLASH_Invalid-Argument | An invalid argument is provided. |
| kStatus_FLASH_Erase-KeyError | API erase key is invalid. |
| kStatus_FLASH_Execute-InRamFunctionNotReady | Execute-in-RAM function is not available. |
| kStatus_FLASH_Access-Error | Invalid instruction codes and out-of bounds addresses. |

| | |
|---|---|
| *kStatus_FLASH_-ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during command execution. |
| *kStatus_FLASH_Eeprom-SingleBitFault* | EEPROM single bit fault error code. |
| *kStatus_FLASH_Eeprom-DoubleBitFault* | EEPROM double bit fault error code. |

### 9.5.7 status_t FLASH_Program ( flash_config_t ∗ *config*, uint32_t *start*, uint32_t ∗ *src*, uint32_t *lengthInBytes* )

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 word-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 word-aligned. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_-AlignmentError* | Parameter is not aligned with the specified baseline. |
| *kStatus_FLASH_Address-Error* | Address is out of range. |

| | |
|---|---|
| *kStatus_FLASH_ExecuteInRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_AccessError* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_CommandFailure* | Run-time error during the command execution. |

### 9.5.8 status_t FLASH_ProgramOnce ( flash_config_t ∗ *config*, uint32_t *index*, uint32_t ∗ *src*, uint32_t *lengthInBytes* )

This function programs the Program Once Field with the desired data for a given flash area as determined by the index and length.

Parameters

| | |
|---|---|
| *config* | A pointer to the storage for the driver runtime state. |
| *index* | The index indicating which area of the Program Once Field to be programmed. |
| *src* | A pointer to the source buffer of data that is to be programmed into the Program Once Field. |
| *lengthInBytes* | The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_InvalidArgument* | An invalid argument is provided. |
| *kStatus_FLASH_ExecuteInRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_AccessError* | Invalid instruction codes and out-of bounds addresses. |

| | |
|---|---|
| *kStatus_FLASH_-ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during the command execution. |

### 9.5.9 status_t FLASH_ReadOnce ( flash_config_t * *config,* uint32_t *index,* uint32_t * *dst,* uint32_t *lengthInBytes* )

This function reads the read once feild with given index and length.

Parameters

| | |
|---|---|
| *config* | A pointer to the storage for the driver runtime state. |
| *index* | The index indicating the area of program once field to be read. |
| *dst* | A pointer to the destination buffer of data that is used to store data to be read. |
| *lengthInBytes* | The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_Execute-InRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_Access-Error* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_-ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during the command execution. |

### 9.5.10 status_t FLASH_GetSecurityState ( flash_config_t * *config,* flash_security_state_t * *state* )

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

| | |
|---|---|
| *config* | A pointer to storage for the driver runtime state. |
| *state* | A pointer to the value returned for the current security status code: |

Return values

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

### 9.5.11 status_t FLASH_SecurityBypass ( flash_config_t * *config*, const uint8_t * *backdoorKey* )

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

| | |
|---|---|
| *config* | A pointer to the storage for the driver runtime state. |
| *backdoorKey* | A pointer to the user buffer containing the backdoor key. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_InvalidArgument* | An invalid argument is provided. |
| *kStatus_FLASH_ExecuteInRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_AccessError* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_CommandFailure* | Run-time error during the command execution. |

### 9.5.12 status_t FLASH_VerifyEraseAll ( flash_config_t * *config*, flash_margin_value_t *margin* )

This function checks whether the flash is erased to the specified read margin level.

Parameters

| | |
|---|---|
| *config* | A pointer to the storage for the driver runtime state. |
| *margin* | Read margin choice. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_Execute-InRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_Access-Error* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_-ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during the command execution. |
| *kStatus_FLASH_Eeprom-SingleBitFault* | EEPROM single bit fault error code. |
| *kStatus_FLASH_Eeprom-DoubleBitFault* | EEPROM double bit fault error code. |

### 9.5.13 status_t FLASH_VerifyErase ( flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, flash_margin_value_t *margin* )

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. |
| *margin* | Read margin choice. |
| *start* | The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned. |

| | |
|---|---|
| *lengthInBytes* | The length, given in bytes (not words or long-words), to be verified. Must be word-aligned. |

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 specified baseline. |
| *kStatus_FLASH_AddressError* | Address is out of range. |
| *kStatus_FLASH_ExecuteInRamFunctionNotReady* | Execute-in-RAM function is not available. |
| *kStatus_FLASH_AccessError* | Invalid instruction codes and out-of bounds addresses. |
| *kStatus_FLASH_ProtectionViolation* | The program/erase operation is requested to execute on protected areas. |
| *kStatus_FLASH_CommandFailure* | Run-time error during the command execution. |

### 9.5.14 status_t FLASH_IsProtected ( flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, flash_protection_state_t * *protection_state* )

This function retrieves the current flash protect status 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 checked. Must be word-aligned. |
| *lengthInBytes* | The length, given in bytes (not words or long-words) to be checked. Must be word-aligned. |
| *protection_state* | A pointer to the value returned for the current protection status code for the desired flash area. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_-AlignmentError* | Parameter is not aligned with specified baseline. |
| *kStatus_FLASH_Address-Error* | The address is out of range. |

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

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. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_-UnknownProperty* | An unknown property tag. |

### 9.5.16 status_t FLASH_SetProperty ( flash_config_t * *config*, flash_property_tag_t *whichProperty*, uint32_t *value* )

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 to set for the desired flash property. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_-UnknownProperty* | An unknown property tag. |
| *kStatus_FLASH_Invalid-PropertyValue* | An invalid property value. |
| *kStatus_FLASH_Read-OnlyProperty* | An read-only property tag. |

### 9.5.17 status_t FLASH_PflashSetProtection ( flash_config_t * *config*, pflash_protection_status_t * *protectStatus* )

Parameters

| | |
|---|---|
| *config* | A pointer to storage for the driver runtime state. |
| *protectStatus* | The expected protect status to set to the PFlash protection register. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_Invalid-Argument* | An invalid argument is provided. |
| *kStatus_FLASH_-CommandFailure* | Run-time error during command execution. |

### 9.5.18 status_t FLASH_PflashGetProtection ( flash_config_t * *config*, pflash_protection_status_t * *protectStatus* )

Parameters

| | |
|---|---|
| *config* | A pointer to the storage for the driver runtime state. |
| *protectStatus* | Protect status returned by the PFlash IP. |

Return values

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

### 9.5.19 status_t FLASH_PflashSetPrefetchSpeculation ( flash_prefetch_speculation_status_t * *speculationStatus* )

Parameters

| | |
|---|---|
| *speculationStatus* | The expected protect status to set to the PFlash protection register. Each bit is |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |
| *kStatus_FLASH_InvalidSpeculationOption* | An invalid speculation option argument is provided. |

### 9.5.20 status_t FLASH_PflashGetPrefetchSpeculation ( flash_prefetch_speculation_status_t * *speculationStatus* )

Parameters

| | |
|---|---|
| *speculationStatus* | Speculation status returned by the PFlash IP. |

Return values

| | |
|---|---|
| *kStatus_FLASH_Success* | API was executed successfully. |

# Chapter 10
# FTM: FlexTimer Driver

## 10.1 Overview

The MCUXpresso SDK provides a driver for the FlexTimer Module (FTM) of MCUXpresso SDK devices.

## 10.2 Function groups

The FTM driver supports the generation of PWM signals, input capture, dual edge capture, output compare, and quadrature decoder modes. The driver also supports configuring each of the FTM fault inputs.

### 10.2.1 Initialization and deinitialization

The function FTM_Init() initializes the FTM with specified configurations. The function FTM_GetDefaultConfig() gets the default configurations. The initialization function configures the FTM for the requested register update mode for registers with buffers. It also sets up the FTM's fault operation mode and FTM behavior in the BDM mode.

The function FTM_Deinit() disables the FTM counter and turns off the module clock.

### 10.2.2 PWM Operations

The function FTM_SetupPwm() sets up FTM channels for the PWM output. The function sets up the PWM signal properties for multiple channels. Each channel has its own duty cycle and level-mode specified. However, the same PWM period and PWM mode is applied to all channels requesting the PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle).

The function FTM_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular FTM channel.

The function FTM_UpdateChnlEdgeLevelSelect() updates the level select bits of a particular FTM channel. This can be used to disable the PWM output when making changes to the PWM signal.

### 10.2.3 Input capture operations

The function FTM_SetupInputCapture() sets up an FTM channel for the input capture. The user can specify the capture edge and a filter value to be used when processing the input signal.

The function FTM_SetupDualEdgeCapture() can be used to measure the pulse width of a signal. A channel pair is used during capture with the input signal coming through a channel n. The user can specify whether to use one-shot or continuous capture, the capture edge for each channel, and any filter value to be used when processing the input signal.

### 10.2.4 Output compare operations

The function FTM_SetupOutputCompare() sets up an FTM channel for the output comparison. The user can specify the channel output on a successful comparison and a comparison value.

### 10.2.5 Quad decode

The function FTM_SetupQuadDecode() sets up FTM channels 0 and 1 for quad decoding. The user can specify the quad decoding mode, polarity, and filter properties for each input signal.

### 10.2.6 Fault operation

The function FTM_SetupFault() sets up the properties for each fault. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

## 10.3 Register Update

Some of the FTM registers have buffers. The driver supports various methods to update these registers with the content of the register buffer. The registers can be updated using the PWM synchronized loading or an intermediate point loading. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ftmMultiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration ftm_pwm_sync_method_t to the pwmSyncMode field.

When using an intermediate reload points, the PWM synchnronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration ftm_reload_point_t to the reloadPoints field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ftm

## 10.4 Typical use case

### 10.4.1 PWM output

Output a PWM signal on two FTM channels with different duty cycles. Periodically update the PWM signal duty cycle. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ftm

## Data Structures

- struct ftm_chnl_pwm_signal_param_t
  *Options to configure a FTM channel's PWM signal. More...*
- struct ftm_chnl_pwm_config_param_t
  *Options to configure a FTM channel using precise setting. More...*
- struct ftm_dual_edge_capture_param_t
  *FlexTimer dual edge capture parameters. More...*
- struct ftm_phase_params_t
  *FlexTimer quadrature decode phase parameters. More...*
- struct ftm_fault_param_t
  *Structure is used to hold the parameters to configure a FTM fault. More...*
- struct ftm_config_t
  *FTM configuration structure. More...*

## Enumerations

- enum ftm_chnl_t {
  kFTM_Chnl_0 = 0U,
  kFTM_Chnl_1,
  kFTM_Chnl_2,
  kFTM_Chnl_3,
  kFTM_Chnl_4,
  kFTM_Chnl_5,
  kFTM_Chnl_6,
  kFTM_Chnl_7 }
  *List of FTM channels.*
- enum ftm_fault_input_t {
  kFTM_Fault_0 = 0U,
  kFTM_Fault_1,
  kFTM_Fault_2,
  kFTM_Fault_3 }
  *List of FTM faults.*
- enum ftm_pwm_mode_t {
  kFTM_EdgeAlignedPwm = 0U,
  kFTM_CenterAlignedPwm,
  kFTM_EdgeAlignedCombinedPwm,
  kFTM_CenterAlignedCombinedPwm,
  kFTM_AsymmetricalCombinedPwm }
  *FTM PWM operation modes.*
- enum ftm_pwm_level_select_t {

kFTM_NoPwmSignal = 0U,
kFTM_LowTrue,
kFTM_HighTrue }

*FTM PWM output pulse mode: high-true, low-true or no output.*

- enum ftm_output_compare_mode_t {
kFTM_NoOutputSignal = (1U << FTM_CnSC_MSA_SHIFT),
kFTM_ToggleOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (1U << FTM_CnSC_ELSA_SHIFT)),
kFTM_ClearOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (2U << FTM_CnSC_ELSA_SHIFT)),
kFTM_SetOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (3U << FTM_CnSC_ELSA_SHIFT)) }

*FlexTimer output compare mode.*

- enum ftm_input_capture_edge_t {
kFTM_RisingEdge = (1U << FTM_CnSC_ELSA_SHIFT),
kFTM_FallingEdge = (2U << FTM_CnSC_ELSA_SHIFT),
kFTM_RiseAndFallEdge = (3U << FTM_CnSC_ELSA_SHIFT) }

*FlexTimer input capture edge.*

- enum ftm_dual_edge_capture_mode_t {
kFTM_OneShot = 0U,
kFTM_Continuous = (1U << FTM_CnSC_MSA_SHIFT) }

*FlexTimer dual edge capture modes.*

- enum ftm_quad_decode_mode_t {
kFTM_QuadPhaseEncode = 0U,
kFTM_QuadCountAndDir }

*FlexTimer quadrature decode modes.*

- enum ftm_phase_polarity_t {
kFTM_QuadPhaseNormal = 0U,
kFTM_QuadPhaseInvert }

*FlexTimer quadrature phase polarities.*

- enum ftm_deadtime_prescale_t {
kFTM_Deadtime_Prescale_1 = 1U,
kFTM_Deadtime_Prescale_4,
kFTM_Deadtime_Prescale_16 }

*FlexTimer pre-scaler factor for the dead time insertion.*

- enum ftm_clock_source_t {
kFTM_SystemClock = 1U,
kFTM_FixedClock,
kFTM_ExternalClock }

*FlexTimer clock source selection.*

- enum ftm_clock_prescale_t {

kFTM_Prescale_Divide_1 = 0U,
kFTM_Prescale_Divide_2,
kFTM_Prescale_Divide_4,
kFTM_Prescale_Divide_8,
kFTM_Prescale_Divide_16,
kFTM_Prescale_Divide_32,
kFTM_Prescale_Divide_64,
kFTM_Prescale_Divide_128 }
*FlexTimer pre-scaler factor selection for the clock source.*
- enum ftm_bdm_mode_t {
kFTM_BdmMode_0 = 0U,
kFTM_BdmMode_1,
kFTM_BdmMode_2,
kFTM_BdmMode_3 }
*Options for the FlexTimer behaviour in BDM Mode.*
- enum ftm_fault_mode_t {
kFTM_Fault_Disable = 0U,
kFTM_Fault_EvenChnls,
kFTM_Fault_AllChnlsMan,
kFTM_Fault_AllChnlsAuto }
*Options for the FTM fault control mode.*
- enum ftm_external_trigger_t {
kFTM_Chnl0Trigger = (1U << 4),
kFTM_Chnl1Trigger = (1U << 5),
kFTM_Chnl2Trigger = (1U << 0),
kFTM_Chnl3Trigger = (1U << 1),
kFTM_Chnl4Trigger = (1U << 2),
kFTM_Chnl5Trigger = (1U << 3),
kFTM_InitTrigger = (1U << 6) }
*FTM external trigger options.*
- enum ftm_pwm_sync_method_t {
kFTM_SoftwareTrigger = FTM_SYNC_SWSYNC_MASK,
kFTM_HardwareTrigger_0 = FTM_SYNC_TRIG0_MASK,
kFTM_HardwareTrigger_1 = FTM_SYNC_TRIG1_MASK,
kFTM_HardwareTrigger_2 = FTM_SYNC_TRIG2_MASK }
*FlexTimer PWM sync options to update registers with buffer.*
- enum ftm_reload_point_t {

kFTM_Chnl0Match = (1U << 0),
kFTM_Chnl1Match = (1U << 1),
kFTM_Chnl2Match = (1U << 2),
kFTM_Chnl3Match = (1U << 3),
kFTM_Chnl4Match = (1U << 4),
kFTM_Chnl5Match = (1U << 5),
kFTM_Chnl6Match = (1U << 6),
kFTM_Chnl7Match = (1U << 7),
kFTM_CntMax = (1U << 8),
kFTM_CntMin = (1U << 9),
kFTM_HalfCycMatch = (1U << 10) }
*FTM options available as loading point for register reload.*
- enum ftm_interrupt_enable_t {
kFTM_Chnl0InterruptEnable = (1U << 0),
kFTM_Chnl1InterruptEnable = (1U << 1),
kFTM_Chnl2InterruptEnable = (1U << 2),
kFTM_Chnl3InterruptEnable = (1U << 3),
kFTM_Chnl4InterruptEnable = (1U << 4),
kFTM_Chnl5InterruptEnable = (1U << 5),
kFTM_Chnl6InterruptEnable = (1U << 6),
kFTM_Chnl7InterruptEnable = (1U << 7),
kFTM_FaultInterruptEnable = (1U << 8),
kFTM_TimeOverflowInterruptEnable = (1U << 9),
kFTM_ReloadInterruptEnable = (1U << 10) }
*List of FTM interrupts.*
- enum ftm_status_flags_t {
kFTM_Chnl0Flag = (1U << 0),
kFTM_Chnl1Flag = (1U << 1),
kFTM_Chnl2Flag = (1U << 2),
kFTM_Chnl3Flag = (1U << 3),
kFTM_Chnl4Flag = (1U << 4),
kFTM_Chnl5Flag = (1U << 5),
kFTM_Chnl6Flag = (1U << 6),
kFTM_Chnl7Flag = (1U << 7),
kFTM_FaultFlag = (1U << 8),
kFTM_TimeOverflowFlag = (1U << 9),
kFTM_ChnlTriggerFlag = (1U << 10),
kFTM_ReloadFlag = (1U << 11) }
*List of FTM flags.*

## Functions

- void FTM_SetupFaultInput (FTM_Type ∗base, ftm_fault_input_t faultNumber, const ftm_fault_param_t ∗faultParams)
*Sets up the working of the FTM fault inputs protection.*
- static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type ∗base, bool enable)

*Enables or disables the FTM global time base signal generation to other FTMs.*
- static void FTM_SetOutputMask (FTM_Type ∗base, ftm_chnl_t chnlNumber, bool mask)
  *Sets the FTM peripheral timer channel output mask.*
- static void FTM_SetSoftwareTrigger (FTM_Type ∗base, bool enable)
  *Enables or disables the FTM software trigger for PWM synchronization.*
- static void FTM_SetWriteProtection (FTM_Type ∗base, bool enable)
  *Enables or disables the FTM write protection.*

## Driver version

- #define FSL_FTM_DRIVER_VERSION (MAKE_VERSION(2, 5, 0))
  *FTM driver version 2.5.0.*

## Initialization and deinitialization

- status_t FTM_Init (FTM_Type ∗base, const ftm_config_t ∗config)
  *Ungates the FTM clock and configures the peripheral for basic operation.*
- void FTM_Deinit (FTM_Type ∗base)
  *Gates the FTM clock.*
- void FTM_GetDefaultConfig (ftm_config_t ∗config)
  *Fills in the FTM configuration structure with the default settings.*
- static ftm_clock_prescale_t FTM_CalculateCounterClkDiv (FTM_Type ∗base, uint32_t counterPeriod_Hz, uint32_t srcClock_Hz)
  *brief Calculates the counter clock prescaler.*

## Channel mode operations

- status_t FTM_SetupPwm (FTM_Type ∗base, const ftm_chnl_pwm_signal_param_t ∗chnlParams, uint8_t numOfChnls, ftm_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
  *Configures the PWM signal parameters.*
- status_t FTM_UpdatePwmDutycycle (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_pwm_mode_t currentPwmMode, uint8_t dutyCyclePercent)
  *Updates the duty cycle of an active PWM signal.*
- void FTM_UpdateChnlEdgeLevelSelect (FTM_Type ∗base, ftm_chnl_t chnlNumber, uint8_t level)
  *Updates the edge level selection for a channel.*
- status_t FTM_SetupPwmMode (FTM_Type ∗base, const ftm_chnl_pwm_config_param_t ∗chnlParams, uint8_t numOfChnls, ftm_pwm_mode_t mode)
  *Configures the PWM mode parameters.*
- void FTM_SetupInputCapture (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_input_capture_edge_t captureMode, uint32_t filterValue)
  *Enables capturing an input signal on the channel using the function parameters.*
- void FTM_SetupOutputCompare (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_output_compare_mode_t compareMode, uint32_t compareValue)
  *Configures the FTM to generate timed pulses.*
- void FTM_SetupDualEdgeCapture (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, const ftm_dual_edge_capture_param_t ∗edgeParam, uint32_t filterValue)
  *Configures the dual edge capture mode of the FTM.*

### Interrupt Interface

- void FTM_EnableInterrupts (FTM_Type ∗base, uint32_t mask)
  *Enables the selected FTM interrupts.*
- void FTM_DisableInterrupts (FTM_Type ∗base, uint32_t mask)
  *Disables the selected FTM interrupts.*
- uint32_t FTM_GetEnabledInterrupts (FTM_Type ∗base)
  *Gets the enabled FTM interrupts.*

### Status Interface

- uint32_t FTM_GetStatusFlags (FTM_Type ∗base)
  *Gets the FTM status flags.*
- void FTM_ClearStatusFlags (FTM_Type ∗base, uint32_t mask)
  *Clears the FTM status flags.*

### Read and write the timer period

- static void FTM_SetTimerPeriod (FTM_Type ∗base, uint32_t ticks)
  *Sets the timer period in units of ticks.*
- static uint32_t FTM_GetCurrentTimerCount (FTM_Type ∗base)
  *Reads the current timer counting value.*
- static uint32_t FTM_GetInputCaptureValue (FTM_Type ∗base, ftm_chnl_t chnlNumber)
  *Reads the captured value.*

### Timer Start and Stop

- static void FTM_StartTimer (FTM_Type ∗base, ftm_clock_source_t clockSource)
  *Starts the FTM counter.*
- static void FTM_StopTimer (FTM_Type ∗base)
  *Stops the FTM counter.*

### Software output control

- static void FTM_SetSoftwareCtrlEnable (FTM_Type ∗base, ftm_chnl_t chnlNumber, bool value)
  *Enables or disables the channel software output control.*
- static void FTM_SetSoftwareCtrlVal (FTM_Type ∗base, ftm_chnl_t chnlNumber, bool value)
  *Sets the channel software output control value.*

### Channel pair operations

- static void FTM_SetFaultControlEnable (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, bool value)
  *This function enables/disables the fault control in a channel pair.*
- static void FTM_SetDeadTimeEnable (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, bool value)
  *This function enables/disables the dead time insertion in a channel pair.*
- static void FTM_SetComplementaryEnable (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, bool value)
  *This function enables/disables complementary mode in a channel pair.*
- static void FTM_SetInvertEnable (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, bool value)
  *This function enables/disables inverting control in a channel pair.*

## Quad Decoder

- void FTM_SetupQuadDecode (FTM_Type *base, const ftm_phase_params_t *phaseAParams, const ftm_phase_params_t *phaseBParams, ftm_quad_decode_mode_t quadMode)
  *Configures the parameters and activates the quadrature decoder mode.*
- static void FTM_SetQuadDecoderModuloValue (FTM_Type *base, uint32_t startValue, uint32_t overValue)
  *Sets the modulo values for Quad Decoder.*
- static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type *base)
  *Gets the current Quad Decoder counter value.*
- static void FTM_ClearQuadDecoderCounterValue (FTM_Type *base)
  *Clears the current Quad Decoder counter value.*

# 10.5 Data Structure Documentation

## 10.5.1 struct ftm_chnl_pwm_signal_param_t

### Data Fields

- ftm_chnl_t chnlNumber
  *The channel/channel pair number.*
- ftm_pwm_level_select_t level
  *PWM output active level select.*
- uint8_t dutyCyclePercent
  *PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)...*
- uint8_t firstEdgeDelayPercent
  *Used only in kFTM_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM.*
- bool enableComplementary
  *Used only in combined PWM mode.*
- bool enableDeadtime
  *Used only in combined PWM mode with enable complementary.*

#### Field Documentation

#### (1) ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber

In combined mode, this represents the channel pair number.

#### (2) ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level

#### (3) uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

#### (4) uint8_t ftm_chnl_pwm_signal_param_t::firstEdgeDelayPercent

Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

#### (5) bool ftm_chnl_pwm_signal_param_t::enableComplementary

true: The combined channels output complementary signals; false: The combined channels output same signals;

#### (6) bool ftm_chnl_pwm_signal_param_t::enableDeadtime

true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

## 10.5.2 struct ftm_chnl_pwm_config_param_t

### Data Fields

- ftm_chnl_t chnlNumber
  *The channel/channel pair number.*
- ftm_pwm_level_select_t level
  *PWM output active level select.*
- uint16_t dutyValue
  *PWM pulse width, the uint of this value is timer ticks.*
- uint16_t firstEdgeValue
  *Used only in kFTM_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM.*
- bool enableComplementary
  *Used only in combined PWM mode.*
- bool enableDeadtime
  *Used only in combined PWM mode with enable complementary.*

### Field Documentation

#### (1) ftm_chnl_t ftm_chnl_pwm_config_param_t::chnlNumber

In combined mode, this represents the channel pair number.

#### (2) ftm_pwm_level_select_t ftm_chnl_pwm_config_param_t::level

#### (3) uint16_t ftm_chnl_pwm_config_param_t::dutyValue

#### (4) uint16_t ftm_chnl_pwm_config_param_t::firstEdgeValue

Specifies the delay to the first edge in a PWM period. If unsure leave as 0, uint of this value is timer ticks.

#### (5) bool ftm_chnl_pwm_config_param_t::enableComplementary

true: The combined channels output complementary signals; false: The combined channels output same signals;

#### (6) bool ftm_chnl_pwm_config_param_t::enableDeadtime

true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

### 10.5.3 struct ftm_dual_edge_capture_param_t

#### Data Fields

- ftm_dual_edge_capture_mode_t mode
  *Dual Edge Capture mode.*
- ftm_input_capture_edge_t currChanEdgeMode
  *Input capture edge select for channel n.*
- ftm_input_capture_edge_t nextChanEdgeMode
  *Input capture edge select for channel n+1.*

### 10.5.4 struct ftm_phase_params_t

#### Data Fields

- bool enablePhaseFilter
  *True: enable phase filter; false: disable filter.*
- uint32_t phaseFilterVal
  *Filter value, used only if phase filter is enabled.*
- ftm_phase_polarity_t phasePolarity
  *Phase polarity.*

### 10.5.5 struct ftm_fault_param_t

#### Data Fields

- bool enableFaultInput
  *True: Fault input is enabled; false: Fault input is disabled.*
- bool faultLevel
  *True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high.*
- bool useFaultFilter
  *True: Use the filtered fault signal; False: Use the direct path from fault input.*

### 10.5.6 struct ftm_config_t

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

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

## Data Fields

- ftm_clock_prescale_t prescale
  *FTM clock prescale value.*
- ftm_bdm_mode_t bdmMode
  *FTM behavior in BDM mode.*
- uint32_t pwmSyncMode
  *Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration ftm_pwm_sync_method_t.*
- uint32_t reloadPoints
  *FTM reload points; When using this, the PWM synchronization is not required.*
- ftm_fault_mode_t faultMode
  *FTM fault control mode.*
- uint8_t faultFilterValue
  *Fault input filter value.*
- ftm_deadtime_prescale_t deadTimePrescale
  *The dead time prescalar value.*
- uint32_t deadTimeValue
  *The dead time value deadTimeValue's available range is 0-1023 when register has DTVALEX, otherwise its available range is 0-63.*
- uint32_t extTriggers
  *External triggers to enable.*
- uint8_t chnlInitState
  *Defines the initialization value of the channels in OUTINT register.*
- uint8_t chnlPolarity
  *Defines the output polarity of the channels in POL register.*
- bool useGlobalTimeBase
  *True: Use of an external global time base is enabled; False: disabled.*

### Field Documentation

#### (1) uint32_t ftm_config_t::pwmSyncMode

#### (2) uint32_t ftm_config_t::reloadPoints

Multiple reload points can be used by providing an OR'ed list of options available in enumeration ftm_reload_point_t.

#### (3) uint32_t ftm_config_t::deadTimeValue

#### (4) uint32_t ftm_config_t::extTriggers

Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration ftm_external_trigger_t.

## 10.6 Macro Definition Documentation

### 10.6.1 #define FSL_FTM_DRIVER_VERSION (MAKE_VERSION(2, 5, 0))

## 10.7 Enumeration Type Documentation

### 10.7.1 enum ftm_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

***kFTM_Chnl_0*** FTM channel number 0.
***kFTM_Chnl_1*** FTM channel number 1.
***kFTM_Chnl_2*** FTM channel number 2.
***kFTM_Chnl_3*** FTM channel number 3.
***kFTM_Chnl_4*** FTM channel number 4.
***kFTM_Chnl_5*** FTM channel number 5.
***kFTM_Chnl_6*** FTM channel number 6.
***kFTM_Chnl_7*** FTM channel number 7.

### 10.7.2 enum ftm_fault_input_t

Enumerator

***kFTM_Fault_0*** FTM fault 0 input pin.
***kFTM_Fault_1*** FTM fault 1 input pin.
***kFTM_Fault_2*** FTM fault 2 input pin.
***kFTM_Fault_3*** FTM fault 3 input pin.

### 10.7.3 enum ftm_pwm_mode_t

Enumerator

***kFTM_EdgeAlignedPwm*** Edge-aligned PWM.
***kFTM_CenterAlignedPwm*** Center-aligned PWM.
***kFTM_EdgeAlignedCombinedPwm*** Edge-aligned combined PWM.
***kFTM_CenterAlignedCombinedPwm*** Center-aligned combined PWM.
***kFTM_AsymmetricalCombinedPwm*** Asymmetrical combined PWM.

### 10.7.4 enum ftm_pwm_level_select_t

Enumerator

***kFTM_NoPwmSignal*** No PWM output on pin.
***kFTM_LowTrue*** Low true pulses.
***kFTM_HighTrue*** High true pulses.

### 10.7.5 enum ftm_output_compare_mode_t

Enumerator

***kFTM_NoOutputSignal*** No channel output when counter reaches CnV.
***kFTM_ToggleOnMatch*** Toggle output.
***kFTM_ClearOnMatch*** Clear output.
***kFTM_SetOnMatch*** Set output.

### 10.7.6 enum ftm_input_capture_edge_t

Enumerator

***kFTM_RisingEdge*** Capture on rising edge only.
***kFTM_FallingEdge*** Capture on falling edge only.
***kFTM_RiseAndFallEdge*** Capture on rising or falling edge.

### 10.7.7 enum ftm_dual_edge_capture_mode_t

Enumerator

***kFTM_OneShot*** One-shot capture mode.
***kFTM_Continuous*** Continuous capture mode.

### 10.7.8 enum ftm_quad_decode_mode_t

Enumerator

***kFTM_QuadPhaseEncode*** Phase A and Phase B encoding mode.
***kFTM_QuadCountAndDir*** Count and direction encoding mode.

### 10.7.9 enum ftm_phase_polarity_t

Enumerator

***kFTM_QuadPhaseNormal*** Phase input signal is not inverted.
***kFTM_QuadPhaseInvert*** Phase input signal is inverted.

### 10.7.10 enum ftm_deadtime_prescale_t

Enumerator

***kFTM_Deadtime_Prescale_1*** Divide by 1.
***kFTM_Deadtime_Prescale_4*** Divide by 4.
***kFTM_Deadtime_Prescale_16*** Divide by 16.

### 10.7.11 enum ftm_clock_source_t

Enumerator

***kFTM_SystemClock*** System clock selected.
***kFTM_FixedClock*** Fixed frequency clock.
***kFTM_ExternalClock*** External clock.

### 10.7.12 enum ftm_clock_prescale_t

Enumerator

***kFTM_Prescale_Divide_1*** Divide by 1.
***kFTM_Prescale_Divide_2*** Divide by 2.
***kFTM_Prescale_Divide_4*** Divide by 4.
***kFTM_Prescale_Divide_8*** Divide by 8.
***kFTM_Prescale_Divide_16*** Divide by 16.
***kFTM_Prescale_Divide_32*** Divide by 32.
***kFTM_Prescale_Divide_64*** Divide by 64.
***kFTM_Prescale_Divide_128*** Divide by 128.

### 10.7.13 enum ftm_bdm_mode_t

Enumerator

***kFTM_BdmMode_0*** FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.

***kFTM_BdmMode_1*** FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
***kFTM_BdmMode_2*** FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
***kFTM_BdmMode_3*** FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

### 10.7.14 enum ftm_fault_mode_t

Enumerator

***kFTM_Fault_Disable*** Fault control is disabled for all channels.
***kFTM_Fault_EvenChnls*** Enabled for even channels only(0,2,4,6) with manual fault clearing.
***kFTM_Fault_AllChnlsMan*** Enabled for all channels with manual fault clearing.
***kFTM_Fault_AllChnlsAuto*** Enabled for all channels with automatic fault clearing.

### 10.7.15 enum ftm_external_trigger_t

Note

Actual available external trigger sources are SoC-specific

Enumerator

***kFTM_Chnl0Trigger*** Generate trigger when counter equals chnl 0 CnV reg.
***kFTM_Chnl1Trigger*** Generate trigger when counter equals chnl 1 CnV reg.
***kFTM_Chnl2Trigger*** Generate trigger when counter equals chnl 2 CnV reg.
***kFTM_Chnl3Trigger*** Generate trigger when counter equals chnl 3 CnV reg.
***kFTM_Chnl4Trigger*** Generate trigger when counter equals chnl 4 CnV reg.
***kFTM_Chnl5Trigger*** Generate trigger when counter equals chnl 5 CnV reg.
***kFTM_InitTrigger*** Generate Trigger when counter is updated with CNTIN.

### 10.7.16 enum ftm_pwm_sync_method_t

Enumerator

***kFTM_SoftwareTrigger*** Software triggers PWM sync.
***kFTM_HardwareTrigger_0*** Hardware trigger 0 causes PWM sync.
***kFTM_HardwareTrigger_1*** Hardware trigger 1 causes PWM sync.
***kFTM_HardwareTrigger_2*** Hardware trigger 2 causes PWM sync.

### 10.7.17 enum ftm_reload_point_t

Note

Actual available reload points are SoC-specific

Enumerator

***kFTM_Chnl0Match*** Channel 0 match included as a reload point.
***kFTM_Chnl1Match*** Channel 1 match included as a reload point.
***kFTM_Chnl2Match*** Channel 2 match included as a reload point.
***kFTM_Chnl3Match*** Channel 3 match included as a reload point.
***kFTM_Chnl4Match*** Channel 4 match included as a reload point.
***kFTM_Chnl5Match*** Channel 5 match included as a reload point.
***kFTM_Chnl6Match*** Channel 6 match included as a reload point.
***kFTM_Chnl7Match*** Channel 7 match included as a reload point.
***kFTM_CntMax*** Use in up-down count mode only, reload when counter reaches the maximum value.

***kFTM_CntMin*** Use in up-down count mode only, reload when counter reaches the minimum value.

***kFTM_HalfCycMatch*** Available on certain SoC's, half cycle match reload point.

### 10.7.18 enum ftm_interrupt_enable_t

Note

Actual available interrupts are SoC-specific

Enumerator

***kFTM_Chnl0InterruptEnable*** Channel 0 interrupt.
***kFTM_Chnl1InterruptEnable*** Channel 1 interrupt.
***kFTM_Chnl2InterruptEnable*** Channel 2 interrupt.
***kFTM_Chnl3InterruptEnable*** Channel 3 interrupt.
***kFTM_Chnl4InterruptEnable*** Channel 4 interrupt.
***kFTM_Chnl5InterruptEnable*** Channel 5 interrupt.
***kFTM_Chnl6InterruptEnable*** Channel 6 interrupt.
***kFTM_Chnl7InterruptEnable*** Channel 7 interrupt.
***kFTM_FaultInterruptEnable*** Fault interrupt.
***kFTM_TimeOverflowInterruptEnable*** Time overflow interrupt.
***kFTM_ReloadInterruptEnable*** Reload interrupt; Available only on certain SoC's.

### 10.7.19 enum ftm_status_flags_t

Note

Actual available flags are SoC-specific

Enumerator

***kFTM_Chnl0Flag*** Channel 0 Flag.
***kFTM_Chnl1Flag*** Channel 1 Flag.
***kFTM_Chnl2Flag*** Channel 2 Flag.
***kFTM_Chnl3Flag*** Channel 3 Flag.
***kFTM_Chnl4Flag*** Channel 4 Flag.
***kFTM_Chnl5Flag*** Channel 5 Flag.
***kFTM_Chnl6Flag*** Channel 6 Flag.
***kFTM_Chnl7Flag*** Channel 7 Flag.
***kFTM_FaultFlag*** Fault Flag.
***kFTM_TimeOverflowFlag*** Time overflow Flag.
***kFTM_ChnlTriggerFlag*** Channel trigger Flag.
***kFTM_ReloadFlag*** Reload Flag; Available only on certain SoC's.

## 10.8 Function Documentation

### 10.8.1 status_t FTM_Init ( FTM_Type ∗ *base*, const ftm_config_t ∗ *config* )

Note

This API should be called at the beginning of the application which is using the FTM driver. If the FTM instance has only TPM features, please use the TPM driver.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *config* | Pointer to the user configuration structure. |

Returns

kStatus_Success indicates success; Else indicates failure.

### 10.8.2 void FTM_Deinit ( FTM_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

### 10.8.3 void FTM_GetDefaultConfig ( ftm_config_t * *config* )

The default values are:

```
*   config->prescale = kFTM_Prescale_Divide_1;
*   config->bdmMode = kFTM_BdmMode_0;
*   config->pwmSyncMode = kFTM_SoftwareTrigger;
*   config->reloadPoints = 0;
*   config->faultMode = kFTM_Fault_Disable;
*   config->faultFilterValue = 0;
*   config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
*   config->deadTimeValue =  0;
*   config->extTriggers = 0;
*   config->chnlInitState = 0;
*   config->chnlPolarity = 0;
*   config->useGlobalTimeBase = false;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the user configuration structure. |

### 10.8.4 static ftm_clock_prescale_t FTM_CalculateCounterClkDiv ( FTM_Type * *base*, uint32_t *counterPeriod_Hz*, uint32_t *srcClock_Hz* ) [inline], [static]

This function calculates the values for SC[PS] bit.

param base FTM peripheral base address param counterPeriod_Hz The desired frequency in Hz which corresponding to the time when the counter reaches the mod value param srcClock_Hz FTM counter clock in Hz

return Calculated clock prescaler value, see ftm_clock_prescale_t.

### 10.8.5 status_t FTM_SetupPwm ( FTM_Type * *base*, const ftm_chnl_pwm_signal-_param_t * *chnlParams*, uint8_t *numOfChnls*, ftm_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz* )

Call this function to configure the PWM signal period, mode, duty cycle, and edge. Use this function to configure all FTM channels that are used to output a PWM signal.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlParams* | Array of PWM channel parameters to configure the channel(s) |
| *numOfChnls* | Number of channels to configure; This should be the size of the array passed in |
| *mode* | PWM operation mode, options available in enumeration ftm_pwm_mode_t |
| *pwmFreq_Hz* | PWM signal frequency in Hz |
| *srcClock_Hz* | FTM counter clock in Hz |

Returns

kStatus_Success if the PWM setup was successful kStatus_Error on failure

### 10.8.6 status_t FTM_UpdatePwmDutycycle ( FTM_Type ∗ *base*, ftm_chnl_t *chnlNumber*, ftm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | The channel/channel pair number. In combined mode, this represents the channel pair number |
| *currentPwm-Mode* | The current PWM mode set during PWM setup |
| *dutyCycle-Percent* | New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle) |

Returns

kStatus_Success if the PWM update was successful kStatus_Error on failure

### 10.8.7 void FTM_UpdateChnlEdgeLevelSelect ( FTM_Type ∗ *base*, ftm_chnl_t *chnlNumber*, uint8_t *level* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | The channel number |
| *level* | The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field. |

### 10.8.8 status_t FTM_SetupPwmMode ( FTM_Type * *base*, const ftm_chnl_pwm_config_param_t * *chnlParams*, uint8_t *numOfChnls*, ftm_pwm_mode_t *mode* )

Call this function to configure the PWM signal mode, duty cycle in ticks, and edge. Use this function to configure all FTM channels that are used to output a PWM signal. Please note that: This API is similar with FTM_SetupPwm() API, but will not set the timer period, and this API will set channel match value in timer ticks, not period percent.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlParams* | Array of PWM channel parameters to configure the channel(s) |
| *numOfChnls* | Number of channels to configure; This should be the size of the array passed in |
| *mode* | PWM operation mode, options available in enumeration ftm_pwm_mode_t |

Returns

kStatus_Success if the PWM setup was successful kStatus_Error on failure

### 10.8.9 void FTM_SetupInputCapture ( FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_input_capture_edge_t *captureMode*, uint32_t *filterValue* )

When the edge specified in the captureMode argument occurs on the channel, the FTM counter is captured into the CnV register. The user has to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only for channels 0, 1, 2, 3.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | The channel number |
| *captureMode* | Specifies which edge to capture |
| *filterValue* | Filter value, specify 0 to disable filter. Available only for channels 0-3. |

### 10.8.10 void FTM_SetupOutputCompare ( FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_output_compare_mode_t *compareMode*, uint32_t *compareValue* )

When the FTM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | The channel number |
| *compareMode* | Action to take on the channel output when the compare condition is met |
| *compareValue* | Value to be programmed in the CnV register. |

### 10.8.11 void FTM_SetupDualEdgeCapture ( FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, const ftm_dual_edge_capture_param_t * *edgeParam*, uint32_t *filterValue* )

This function sets up the dual edge capture mode on a channel pair. The capture edge for the channel pair and the capture mode (one-shot or continuous) is specified in the parameter argument. The filter function is disabled if the filterVal argument passed is zero. The filter function is available only on channels 0 and 2. The user has to read the channel CnV registers separately to get the capture values.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlPair-Number* | The FTM channel pair number; options are 0, 1, 2, 3 |

| | |
|---|---|
| *edgeParam* | Sets up the dual edge capture function |
| *filterValue* | Filter value, specify 0 to disable filter. Available only for channel pair 0 and 1. |

### 10.8.12 void FTM_SetupFaultInput ( FTM_Type * *base*, ftm_fault_input_t *faultNumber*, const ftm_fault_param_t * *faultParams* )

FTM can have up to 4 fault inputs. This function sets up fault parameters, fault level, and input filter.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *faultNumber* | FTM fault to configure. |
| *faultParams* | Parameters passed in to set up the fault |

### 10.8.13 void FTM_EnableInterrupts ( FTM_Type * *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration ftm_-interrupt_enable_t |

### 10.8.14 void FTM_DisableInterrupts ( FTM_Type * *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration ftm_-interrupt_enable_t |

### 10.8.15 uint32_t FTM_GetEnabledInterrupts ( FTM_Type * *base* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

Returns

The enabled interrupts. This is the logical OR of members of the enumeration ftm_interrupt_enable-_t

### 10.8.16 uint32_t FTM_GetStatusFlags ( FTM_Type * *base* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

Returns

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

### 10.8.17 void FTM_ClearStatusFlags ( FTM_Type * *base,* uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *mask* | The status flags to clear. This is a logical OR of members of the enumeration ftm_-status_flags_t |

### 10.8.18 static void FTM_SetTimerPeriod ( FTM_Type * *base,* uint32_t *ticks* ) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
2. Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *ticks* | A timer period in units of ticks, which should be equal or greater than 1. |

### 10.8.19 static uint32_t FTM_GetCurrentTimerCount ( FTM_Type * *base* ) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

Returns

The current counter value in ticks

### 10.8.20 static uint32_t FTM_GetInputCaptureValue ( FTM_Type * *base*, ftm_chnl_t *chnlNumber* ) [inline], [static]

This function returns the captured value of a FTM channel configured in input capture or dual edge capture mode.

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

| | |
|---|---|
| *chnlNumber* | Channel to be read |

Returns

The captured FTM counter value of the input modes.

### 10.8.21 static void FTM_StartTimer ( FTM_Type * *base*, ftm_clock_source_t *clockSource* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *clockSource* | FTM clock source; After the clock source is set, the counter starts running. |

### 10.8.22 static void FTM_StopTimer ( FTM_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |

### 10.8.23 static void FTM_SetSoftwareCtrlEnable ( FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | Channel to be enabled or disabled |
| *value* | true: channel output is affected by software output control false: channel output is unaffected by software output control |

### 10.8.24 static void FTM_SetSoftwareCtrlVal ( FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address. |
| *chnlNumber* | Channel to be configured |
| *value* | true to set 1, false to set 0 |

### 10.8.25 static void FTM_SetGlobalTimeBaseOutputEnable ( FTM_Type * *base,* bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *enable* | true to enable, false to disable |

### 10.8.26 static void FTM_SetOutputMask ( FTM_Type * *base,* ftm_chnl_t *chnlNumber,* bool *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlNumber* | Channel to be configured |
| *mask* | true: masked, channel is forced to its inactive state; false: unmasked |

### 10.8.27 static void FTM_SetFaultControlEnable ( FTM_Type * *base,* ftm_chnl_t *chnlPairNumber,* bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlPair-Number* | The FTM channel pair number; options are 0, 1, 2, 3 |

| | |
|---|---|
| *value* | true: Enable fault control for this channel pair; false: No fault control |

### 10.8.28 static void FTM_SetDeadTimeEnable ( FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlPair-Number* | The FTM channel pair number; options are 0, 1, 2, 3 |
| *value* | true: Insert dead time in this channel pair; false: No dead time inserted |

### 10.8.29 static void FTM_SetComplementaryEnable ( FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlPair-Number* | The FTM channel pair number; options are 0, 1, 2, 3 |
| *value* | true: enable complementary mode; false: disable complementary mode |

### 10.8.30 static void FTM_SetInvertEnable ( FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *chnlPair-Number* | The FTM channel pair number; options are 0, 1, 2, 3 |

| | |
|---|---|
| *value* | true: enable inverting; false: disable inverting |

### 10.8.31 void FTM_SetupQuadDecode ( FTM_Type * *base*, const ftm_phase_params_t * *phaseAParams*, const ftm_phase_params_t * *phaseBParams*, ftm_quad_decode_mode_t *quadMode* )

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *phaseAParams* | Phase A configuration parameters |
| *phaseBParams* | Phase B configuration parameters |
| *quadMode* | Selects encoding mode used in quadrature decoder mode |

### 10.8.32 static void FTM_SetQuadDecoderModuloValue ( FTM_Type * *base*, uint32_t *startValue*, uint32_t *overValue* ) [inline], [static]

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address. |
| *startValue* | The low limit value for Quad Decoder counter. |
| *overValue* | The high limit value for Quad Decoder counter. |

### 10.8.33 static uint32_t FTM_GetQuadDecoderCounterValue ( FTM_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address. |

Returns

Current quad Decoder counter value.

### 10.8.34 static void FTM_ClearQuadDecoderCounterValue ( FTM_Type * *base* ) [inline], [static]

The counter is set as the initial value.

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address. |

### 10.8.35 static void FTM_SetSoftwareTrigger ( FTM_Type * *base,* bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *enable* | true: software trigger is selected, false: software trigger is not selected |

### 10.8.36 static void FTM_SetWriteProtection ( FTM_Type * *base,* bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | FTM peripheral base address |
| *enable* | true: Write-protection is enabled, false: Write-protection is disabled |

# Chapter 11
# GPIO: General-Purpose Input/Output Driver

## 11.1 Overview

### Modules

- FGPIO Driver
- GPIO Driver

### Data Structures

- struct gpio_pin_config_t
  *The GPIO pin configuration structure. More...*

### Enumerations

- enum gpio_port_num_t
  *PORT definition.*
- enum gpio_pin_direction_t {
  kGPIO_DigitalInput = 0U,
  kGPIO_DigitalOutput = 1U }
  *GPIO direction definition.*

### Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))
  *GPIO driver version.*

## 11.2 Data Structure Documentation

### 11.2.1 struct gpio_pin_config_t

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

### Data Fields

- gpio_pin_direction_t pinDirection
  *GPIO direction, input or output.*
- uint8_t outputLogic
  *Set a default output logic, which has no use in input.*

## 11.3 Macro Definition Documentation

### 11.3.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

## 11.4 Enumeration Type Documentation

### 11.4.1 enum gpio_pin_direction_t

Enumerator

***kGPIO_DigitalInput*** Set current pin as digital input.
***kGPIO_DigitalOutput*** Set current pin as digital output.

## 11.5 GPIO Driver

### 11.5.1 Overview

The MCUXpresso SDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of MCUXpresso SDK devices.

### 11.5.2 Typical use case

#### 11.5.2.1 Output Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/gpio

#### 11.5.2.2 Input Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/gpio

### GPIO Configuration

- void GPIO_PinInit (gpio_port_num_t port, uint8_t pin, const gpio_pin_config_t *config)
  *Initializes a GPIO pin used by the board.*

### GPIO Output Operations

- void GPIO_PinWrite (gpio_port_num_t port, uint8_t pin, uint8_t output)
  *Sets the output level of the multiple GPIO pins to the logic 1 or 0.*
- void GPIO_PortSet (gpio_port_num_t port, uint8_t mask)
  *Sets the output level of the multiple GPIO pins to the logic 1.*
- void GPIO_PortClear (gpio_port_num_t port, uint8_t mask)
  *Sets the output level of the multiple GPIO pins to the logic 0.*
- void GPIO_PortToggle (gpio_port_num_t port, uint8_t mask)
  *Reverses the current output logic of the multiple GPIO pins.*

### GPIO Input Operations

- uint32_t GPIO_PinRead (gpio_port_num_t port, uint8_t pin)
  *Reads the current input value of the GPIO port.*

### 11.5.3 Function Documentation

#### 11.5.3.1 void GPIO_PinInit ( gpio_port_num_t *port*, uint8_t *pin*, const gpio_pin_config_t ∗ *config* )

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or an 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

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers --— PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. GPIOB serial registers --— PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |
| *pin* | GPIO port pin number |
| *config* | GPIO pin configuration pointer |

#### 11.5.3.2 void GPIO_PinWrite ( gpio_port_num_t *port*, uint8_t *pin*, uint8_t *output* )

Parameters

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers --— PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. GPIOB serial registers --— PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |

| | |
|---|---|
| *pin* | GPIO pin number |
| *output* | GPIO pin output logic level.<br>• 0: corresponding pin output low-logic level.<br>• 1: corresponding pin output high-logic level. |

#### 11.5.3.3 void GPIO_PortSet ( gpio_port_num_t *port,* uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers ---– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. GPIOB serial registers ---– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |
| *mask* | GPIO pin number macro |

#### 11.5.3.4 void GPIO_PortClear ( gpio_port_num_t *port,* uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers ---– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. GPIOB serial registers ---– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |
| *mask* | GPIO pin number macro |

#### 11.5.3.5 void GPIO_PortToggle ( gpio_port_num_t *port,* uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers ---– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. GPIOB serial registers ---– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |

| | |
|---|---|
| *mask* | GPIO pin number macro |

#### 11.5.3.6 uint32_t GPIO_PinRead ( gpio_port_num_t *port*, uint8_t *pin* )

Parameters

| | |
|---|---|
| *port* | GPIO PORT number, see "gpio_port_num_t". For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers -—– PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. GPIOB serial registers -—– PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ... |
| *pin* | GPIO pin number |

Return values

| | |
|---|---|
| *GPIO* | port input value<br>• 0: corresponding pin input low-logic level.<br>• 1: corresponding pin input high-logic level. |

# 11.6 FGPIO Driver

## 11.6.1 Overview

This section describes the programming interface of the FGPIO driver. The FGPIO driver configures the FGPIO module and provides a functional interface to build the GPIO application.

Note

> FGPIO (Fast GPIO) is only available in a few MCUs. FGPIO and GPIO share the same peripheral but use different registers. FGPIO is closer to the core than the regular GPIO and it's faster to read and write.

## 11.6.2 Typical use case

### 11.6.2.1 Output Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/gpio

### 11.6.2.2 Input Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/gpio

## FGPIO Configuration

- void FGPIO_PortInit (gpio_port_num_t port)
  *Initializes the FGPIO peripheral.*
- void FGPIO_PinInit (gpio_port_num_t port, uint8_t pin, const gpio_pin_config_t *config)
  *Initializes a FGPIO pin used by the board.*

## FGPIO Output Operations

- void FGPIO_PinWrite (gpio_port_num_t port, uint8_t pin, uint8_t output)
  *Sets the output level of the multiple FGPIO pins to the logic 1 or 0.*
- void FGPIO_PortSet (gpio_port_num_t port, uint8_t mask)
  *Sets the output level of the multiple FGPIO pins to the logic 1.*
- void FGPIO_PortClear (gpio_port_num_t port, uint8_t mask)
  *Sets the output level of the multiple FGPIO pins to the logic 0.*
- void FGPIO_PortToggle (gpio_port_num_t port, uint8_t mask)
  *Reverses the current output logic of the multiple FGPIO pins.*

## FGPIO Input Operations

- uint32_t FGPIO_PinRead (gpio_port_num_t port, uint8_t pin)

*Reads the current input value of the FGPIO port.*

### 11.6.3 Function Documentation

#### 11.6.3.1 void FGPIO_PortInit ( gpio_port_num_t *port* )

This function ungates the FGPIO clock.

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers --— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers --— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ... |

#### 11.6.3.2 void FGPIO_PinInit ( gpio_port_num_t *port,* uint8_t *pin,* const gpio_pin_config_t ∗ *config* )

To initialize the FGPIO driver, define a pin configuration, as either input or output, in the user file. Then, call the FGPIO_PinInit() function.

This is an example to define an input pin or an 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

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers --— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers --— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ... |

| | |
|---|---|
| *pin* | FGPIO port pin number |
| *config* | FGPIO pin configuration pointer |

#### 11.6.3.3 void FGPIO_PinWrite ( gpio_port_num_t *port,* uint8_t *pin,* uint8_t *output* )

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers -—– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. FGPIOB serial registers -—– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |
| *pin* | FGPIO pin number |
| *output* | FGPIOpin output logic level.<br>• 0: corresponding pin output low-logic level.<br>• 1: corresponding pin output high-logic level. |

#### 11.6.3.4 void FGPIO_PortSet ( gpio_port_num_t *port,* uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers -—– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. FGPIOB serial registers -—– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |
| *mask* | FGPIO pin number macro |

#### 11.6.3.5 void FGPIO_PortClear ( gpio_port_num_t *port,* uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers -—– PTA 0 ∼ 7, PTB 0 ∼7 ... PTD 0 ∼ 7. FGPIOB serial registers -—– PTE 0 ∼ 7, PTF 0 ∼7 ... PTH 0 ∼ 7. ... |

| | |
|---|---|
| *mask* | FGPIO pin number macro |

#### 11.6.3.6 void FGPIO_PortToggle ( gpio_port_num_t *port*, uint8_t *mask* )

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers --— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers --— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ... |
| *mask* | FGPIO pin number macro |

#### 11.6.3.7 uint32_t FGPIO_PinRead ( gpio_port_num_t *port*, uint8_t *pin* )

Parameters

| | |
|---|---|
| *port* | FGPIO PORT number, see "gpio_port_num_t". For each group FGPIO (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers --— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers --— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ... |
| *pin* | FGPIO pin number |

Return values

| | |
|---|---|
| *FGPIO* | port input value<br>• 0: corresponding pin input low-logic level.<br>• 1: corresponding pin input high-logic level. |

# Chapter 12
# I2C: Inter-Integrated Circuit Driver

## 12.1 Overview

### Modules

- I2C CMSIS Driver
- I2C Driver

# 12.2 I2C Driver

## 12.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of MCUXpresso SDK devices.

The I2C driver includes functional APIs and transactional APIs.

Functional APIs target the low-level APIs. Functional APIs can be used for the I2C master/slave initialization/configuration/operation for optimization/customization purpose. Using the functional APIs requires knowing the I2C master peripheral and how to organize functional APIs to meet the application requirements. The I2C functional operation groups provide the functional APIs set.

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions I2C_MasterTransferNonBlocking() set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

## 12.2.2 Typical use case

### 12.2.2.1 Master Operation in functional method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/i2c

### 12.2.2.2 Master Operation in interrupt transactional method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/i2c

### 12.2.2.3 Master Operation in DMA transactional method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/i2c

### 12.2.2.4 Slave Operation in functional method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/i2c

### 12.2.2.5 Slave Operation in interrupt transactional method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/i2c

## Data Structures

- struct i2c_master_config_t
  *I2C master user configuration. More...*
- struct i2c_slave_config_t
  *I2C slave user configuration. More...*
- struct i2c_master_transfer_t
  *I2C master transfer structure. More...*
- struct i2c_master_handle_t
  *I2C master handle structure. More...*
- struct i2c_slave_transfer_t
  *I2C slave transfer structure. More...*
- struct i2c_slave_handle_t
  *I2C slave handle structure. More...*

## Macros

- #define I2C_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/
  *Retry times for waiting flag.*
- #define I2C_MASTER_FACK_CONTROL 0U /∗ Default defines to zero means master will send ack automatically. ∗/
  *Mater Fast ack control, control if master needs to manually write ack, this is used to low the speed of transfer for SoCs with feature FSL_FEATURE_I2C_HAS_DOUBLE_BUFFERING.*

## Typedefs

- typedef void(∗ i2c_master_transfer_callback_t )(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t status, void ∗userData)
  *I2C master transfer callback typedef.*
- typedef void(∗ i2c_slave_transfer_callback_t )(I2C_Type ∗base, i2c_slave_transfer_t ∗xfer, void ∗userData)
  *I2C slave transfer callback typedef.*

## Enumerations

- enum {
  kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_I2C, 0),
  kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_I2C, 1),
  kStatus_I2C_Nak = MAKE_STATUS(kStatusGroup_I2C, 2),
  kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_I2C, 3),
  kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_I2C, 4),
  kStatus_I2C_Addr_Nak = MAKE_STATUS(kStatusGroup_I2C, 5) }
  *I2C status return codes.*

- enum _i2c_flags {
  kI2C_ReceiveNakFlag = I2C_S_RXAK_MASK,
  kI2C_IntPendingFlag = I2C_S_IICIF_MASK,
  kI2C_TransferDirectionFlag = I2C_S_SRW_MASK,
  kI2C_RangeAddressMatchFlag = I2C_S_RAM_MASK,
  kI2C_ArbitrationLostFlag = I2C_S_ARBL_MASK,
  kI2C_BusBusyFlag = I2C_S_BUSY_MASK,
  kI2C_AddressMatchFlag = I2C_S_IAAS_MASK,
  kI2C_TransferCompleteFlag = I2C_S_TCF_MASK,
  kI2C_StopDetectFlag = I2C_FLT_STOPF_MASK << 8,
  kI2C_StartDetectFlag = I2C_FLT_STARTF_MASK << 8 }
  *I2C peripheral flags.*
- enum _i2c_interrupt_enable {
  kI2C_GlobalInterruptEnable = I2C_C1_IICIE_MASK,
  kI2C_StartStopDetectInterruptEnable = I2C_FLT_SSIE_MASK }
  *I2C feature interrupt source.*
- enum i2c_direction_t {
  kI2C_Write = 0x0U,
  kI2C_Read = 0x1U }
  *The direction of master and slave transfers.*
- enum i2c_slave_address_mode_t {
  kI2C_Address7bit = 0x0U,
  kI2C_RangeMatch = 0X2U }
  *Addressing mode.*
- enum _i2c_master_transfer_flags {
  kI2C_TransferDefaultFlag = 0x0U,
  kI2C_TransferNoStartFlag = 0x1U,
  kI2C_TransferRepeatedStartFlag = 0x2U,
  kI2C_TransferNoStopFlag = 0x4U }
  *I2C transfer control flag.*
- enum i2c_slave_transfer_event_t {
  kI2C_SlaveAddressMatchEvent = 0x01U,
  kI2C_SlaveTransmitEvent = 0x02U,
  kI2C_SlaveReceiveEvent = 0x04U,
  kI2C_SlaveTransmitAckEvent = 0x08U,
  kI2C_SlaveStartEvent = 0x10U,
  kI2C_SlaveCompletionEvent = 0x20U,
  kI2C_SlaveGenaralcallEvent = 0x40U,
  kI2C_SlaveAllEvents }
  *Set of events sent to the callback for nonblocking slave transfers.*
- enum { kClearFlags = kI2C_ArbitrationLostFlag | kI2C_IntPendingFlag | kI2C_StartDetectFlag | kI2C_StopDetectFlag }
  *Common sets of flags used by the driver.*

## Driver version

- #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 9))
  *I2C driver version.*

## Initialization and deinitialization

- void I2C_MasterInit (I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  *Initializes the I2C peripheral.*
- void I2C_SlaveInit (I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t srcClock_Hz)
  *Initializes the I2C peripheral.*
- void I2C_MasterDeinit (I2C_Type *base)
  *De-initializes the I2C master peripheral.*
- void I2C_SlaveDeinit (I2C_Type *base)
  *De-initializes the I2C slave peripheral.*
- uint32_t I2C_GetInstance (I2C_Type *base)
  *Get instance number for I2C module.*
- void I2C_MasterGetDefaultConfig (i2c_master_config_t *masterConfig)
  *Sets the I2C master configuration structure to default values.*
- void I2C_SlaveGetDefaultConfig (i2c_slave_config_t *slaveConfig)
  *Sets the I2C slave configuration structure to default values.*
- static void I2C_Enable (I2C_Type *base, bool enable)
  *Enables or disables the I2C peripheral operation.*

## Status

- uint32_t I2C_MasterGetStatusFlags (I2C_Type *base)
  *Gets the I2C status flags.*
- static uint32_t I2C_SlaveGetStatusFlags (I2C_Type *base)
  *Gets the I2C status flags.*
- static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  *Clears the I2C status flag state.*
- static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  *Clears the I2C status flag state.*

## Interrupts

- void I2C_EnableInterrupts (I2C_Type *base, uint32_t mask)
  *Enables I2C interrupt requests.*
- void I2C_DisableInterrupts (I2C_Type *base, uint32_t mask)
  *Disables I2C interrupt requests.*

## DMA Control

- static uint32_t I2C_GetDataRegAddr (I2C_Type ∗base)
  *Gets the I2C tx/rx data register address.*

## Bus Operations

- void I2C_MasterSetBaudRate (I2C_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  *Sets the I2C master transfer baud rate.*
- status_t I2C_MasterStart (I2C_Type ∗base, uint8_t address, i2c_direction_t direction)
  *Sends a START on the I2C bus.*
- status_t I2C_MasterStop (I2C_Type ∗base)
  *Sends a STOP signal on the I2C bus.*
- status_t I2C_MasterRepeatedStart (I2C_Type ∗base, uint8_t address, i2c_direction_t direction)
  *Sends a REPEATED START on the I2C bus.*
- status_t I2C_MasterWriteBlocking (I2C_Type ∗base, const uint8_t ∗txBuff, size_t txSize, uint32_t flags)
  *Performs a polling send transaction on the I2C bus.*
- status_t I2C_MasterReadBlocking (I2C_Type ∗base, uint8_t ∗rxBuff, size_t rxSize, uint32_t flags)
  *Performs a polling receive transaction on the I2C bus.*
- status_t I2C_SlaveWriteBlocking (I2C_Type ∗base, const uint8_t ∗txBuff, size_t txSize)
  *Performs a polling send transaction on the I2C bus.*
- status_t I2C_SlaveReadBlocking (I2C_Type ∗base, uint8_t ∗rxBuff, size_t rxSize)
  *Performs a polling receive transaction on the I2C bus.*
- status_t I2C_MasterTransferBlocking (I2C_Type ∗base, i2c_master_transfer_t ∗xfer)
  *Performs a master polling transfer on the I2C bus.*

## Transactional

- void I2C_MasterTransferCreateHandle (I2C_Type ∗base, i2c_master_handle_t ∗handle, i2c_master_transfer_callback_t callback, void ∗userData)
  *Initializes the I2C handle which is used in transactional functions.*
- status_t I2C_MasterTransferNonBlocking (I2C_Type ∗base, i2c_master_handle_t ∗handle, i2c_master_transfer_t ∗xfer)
  *Performs a master interrupt non-blocking transfer on the I2C bus.*
- status_t I2C_MasterTransferGetCount (I2C_Type ∗base, i2c_master_handle_t ∗handle, size_t ∗count)
  *Gets the master transfer status during a interrupt non-blocking transfer.*
- status_t I2C_MasterTransferAbort (I2C_Type ∗base, i2c_master_handle_t ∗handle)
  *Aborts an interrupt non-blocking transfer early.*
- void I2C_MasterTransferHandleIRQ (I2C_Type ∗base, void ∗i2cHandle)
  *Master interrupt handler.*
- void I2C_SlaveTransferCreateHandle (I2C_Type ∗base, i2c_slave_handle_t ∗handle, i2c_slave_transfer_callback_t callback, void ∗userData)
  *Initializes the I2C handle which is used in transactional functions.*
- status_t I2C_SlaveTransferNonBlocking (I2C_Type ∗base, i2c_slave_handle_t ∗handle, uint32_t eventMask)

*Starts accepting slave transfers.*
- void I2C_SlaveTransferAbort (I2C_Type ∗base, i2c_slave_handle_t ∗handle)
  *Aborts the slave transfer.*
- 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.*
- void I2C_SlaveTransferHandleIRQ (I2C_Type ∗base, void ∗i2cHandle)
  *Slave interrupt handler.*

## 12.2.3 Data Structure Documentation

### 12.2.3.1 struct i2c_master_config_t

#### Data Fields

- bool enableMaster
  *Enables the I2C peripheral at initialization time.*
- bool enableStopHold
  *Controls the stop hold enable.*
- uint32_t baudRate_Bps
  *Baud rate configuration of I2C peripheral.*
- uint8_t glitchFilterWidth
  *Controls the width of the glitch.*

#### Field Documentation

**(1) bool i2c_master_config_t::enableMaster**

**(2) bool i2c_master_config_t::enableStopHold**

**(3) uint32_t i2c_master_config_t::baudRate_Bps**

**(4) uint8_t i2c_master_config_t::glitchFilterWidth**

### 12.2.3.2 struct i2c_slave_config_t

#### Data Fields

- bool enableSlave
  *Enables the I2C peripheral at initialization time.*
- bool enableGeneralCall
  *Enables the general call addressing mode.*
- bool enableWakeUp
  *Enables/disables waking up MCU from low-power mode.*
- bool enableBaudRateCtl
  *Enables/disables independent slave baud rate on SCL in very fast I2C modes.*
- uint16_t slaveAddress
  *A slave address configuration.*
- uint16_t upperAddress
  *A maximum boundary slave address used in a range matching mode.*

- i2c_slave_address_mode_t addressingMode
  *An addressing mode configuration of i2c_slave_address_mode_config_t.*
- uint32_t sclStopHoldTime_ns
  *the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.*

#### Field Documentation

##### (1) bool i2c_slave_config_t::enableSlave

##### (2) bool i2c_slave_config_t::enableGeneralCall

##### (3) bool i2c_slave_config_t::enableWakeUp

##### (4) bool i2c_slave_config_t::enableBaudRateCtl

##### (5) uint16_t i2c_slave_config_t::slaveAddress

##### (6) uint16_t i2c_slave_config_t::upperAddress

##### (7) i2c_slave_address_mode_t i2c_slave_config_t::addressingMode

##### (8) uint32_t i2c_slave_config_t::sclStopHoldTime_ns

### 12.2.3.3 struct i2c_master_transfer_t

#### Data Fields

- uint32_t flags
  *A transfer flag which controls the transfer.*
- uint8_t slaveAddress
  *7-bit slave address.*
- i2c_direction_t direction
  *A transfer direction, read or write.*
- uint32_t subaddress
  *A sub address.*
- uint8_t subaddressSize
  *A size of the command buffer.*
- uint8_t ∗volatile data
  *A transfer buffer.*
- volatile size_t dataSize
  *A transfer size.*

#### Field Documentation

##### (1) uint32_t i2c_master_transfer_t::flags

##### (2) uint8_t i2c_master_transfer_t::slaveAddress

#### (3) i2c_direction_t i2c_master_transfer_t::direction

#### (4) uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

#### (5) uint8_t i2c_master_transfer_t::subaddressSize

#### (6) uint8_t* volatile i2c_master_transfer_t::data

#### (7) volatile size_t i2c_master_transfer_t::dataSize

### 12.2.3.4 struct _i2c_master_handle

I2C master handle typedef.

#### Data Fields

- i2c_master_transfer_t transfer
  *I2C master transfer copy.*
- size_t transferSize
  *Total bytes to be transferred.*
- uint8_t state
  *A transfer state maintained during transfer.*
- i2c_master_transfer_callback_t completionCallback
  *A callback function called when the transfer is finished.*
- void * userData
  *A callback parameter passed to the callback function.*

#### Field Documentation

#### (1) i2c_master_transfer_t i2c_master_handle_t::transfer

#### (2) size_t i2c_master_handle_t::transferSize

#### (3) uint8_t i2c_master_handle_t::state

#### (4) i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

#### (5) void* i2c_master_handle_t::userData

### 12.2.3.5 struct i2c_slave_transfer_t

#### Data Fields

- i2c_slave_transfer_event_t event
  *A reason that the callback is invoked.*
- uint8_t *volatile data
  *A transfer buffer.*
- volatile size_t dataSize

*A transfer size.*
- status_t completionStatus
  *Success or error code describing how the transfer completed.*
- size_t transferredCount
  *A number of bytes actually transferred since the start or since the last repeated start.*

#### Field Documentation

##### (1) i2c_slave_transfer_event_t i2c_slave_transfer_t::event

##### (2) uint8_t∗ volatile i2c_slave_transfer_t::data

##### (3) volatile size_t i2c_slave_transfer_t::dataSize

##### (4) status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

##### (5) size_t i2c_slave_transfer_t::transferredCount

### 12.2.3.6 struct _i2c_slave_handle

I2C slave handle typedef.

#### Data Fields

- volatile bool isBusy
  *Indicates whether a transfer is busy.*
- i2c_slave_transfer_t transfer
  *I2C slave transfer copy.*
- uint32_t eventMask
  *A mask of enabled events.*
- i2c_slave_transfer_callback_t callback
  *A callback function called at the transfer event.*
- void ∗ userData
  *A callback parameter passed to the callback.*

#### Field Documentation

##### (1) volatile bool i2c_slave_handle_t::isBusy

##### (2) i2c_slave_transfer_t i2c_slave_handle_t::transfer

##### (3) uint32_t i2c_slave_handle_t::eventMask

##### (4) i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

##### (5) void∗ i2c_slave_handle_t::userData

### 12.2.4 Macro Definition Documentation

#### 12.2.4.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 9))

#### 12.2.4.2 #define I2C_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/

### 12.2.5 Typedef Documentation

#### 12.2.5.1 typedef void(∗ i2c_master_transfer_callback_t)(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t status, void ∗userData)

#### 12.2.5.2 typedef void(∗ i2c_slave_transfer_callback_t)(I2C_Type ∗base, i2c_slave_transfer_t ∗xfer, void ∗userData)

### 12.2.6 Enumeration Type Documentation

#### 12.2.6.1 anonymous enum

Enumerator

***kStatus_I2C_Busy*** I2C is busy with current transfer.
***kStatus_I2C_Idle*** Bus is Idle.
***kStatus_I2C_Nak*** NAK received during transfer.
***kStatus_I2C_ArbitrationLost*** Arbitration lost during transfer.
***kStatus_I2C_Timeout*** Timeout polling status flags.
***kStatus_I2C_Addr_Nak*** NAK received during the address probe.

#### 12.2.6.2 enum _i2c_flags

Note

These enumerations are meant to be OR'd together to form a bit mask.

Enumerator

***kI2C_ReceiveNakFlag*** I2C receive NAK flag.
***kI2C_IntPendingFlag*** I2C interrupt pending flag. This flag can be cleared.
***kI2C_TransferDirectionFlag*** I2C transfer direction flag.
***kI2C_RangeAddressMatchFlag*** I2C range address match flag.
***kI2C_ArbitrationLostFlag*** I2C arbitration lost flag. This flag can be cleared.
***kI2C_BusBusyFlag*** I2C bus busy flag.
***kI2C_AddressMatchFlag*** I2C address match flag.
***kI2C_TransferCompleteFlag*** I2C transfer complete flag.

***kI2C_StopDetectFlag*** I2C stop detect flag. This flag can be cleared.
***kI2C_StartDetectFlag*** I2C start detect flag. This flag can be cleared.

#### 12.2.6.3 enum _i2c_interrupt_enable

Enumerator

***kI2C_GlobalInterruptEnable*** I2C global interrupt.
***kI2C_StartStopDetectInterruptEnable*** I2C start&stop detect interrupt.

#### 12.2.6.4 enum i2c_direction_t

Enumerator

***kI2C_Write*** Master transmits to the slave.
***kI2C_Read*** Master receives from the slave.

#### 12.2.6.5 enum i2c_slave_address_mode_t

Enumerator

***kI2C_Address7bit*** 7-bit addressing mode.
***kI2C_RangeMatch*** Range address match addressing mode.

#### 12.2.6.6 enum _i2c_master_transfer_flags

Enumerator

***kI2C_TransferDefaultFlag*** A transfer starts with a start signal, stops with a stop signal.
***kI2C_TransferNoStartFlag*** A transfer starts without a start signal, only support write only or write+read with no start flag, do not support read only with no start flag.
***kI2C_TransferRepeatedStartFlag*** A transfer starts with a repeated start signal.
***kI2C_TransferNoStopFlag*** A transfer ends without a stop signal.

#### 12.2.6.7 enum i2c_slave_transfer_event_t

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I2C_SlaveTransferNonBlocking() 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.

Enumerator

***kI2C_SlaveAddressMatchEvent*** Received the slave address after a start or repeated start.
***kI2C_SlaveTransmitEvent*** A callback is requested to provide data to transmit (slave-transmitter role).
***kI2C_SlaveReceiveEvent*** A callback is requested to provide a buffer in which to place received data (slave-receiver role).
***kI2C_SlaveTransmitAckEvent*** A callback needs to either transmit an ACK or NACK.
***kI2C_SlaveStartEvent*** A start/repeated start was detected.
***kI2C_SlaveCompletionEvent*** A stop was detected or finished transfer, completing the transfer.
***kI2C_SlaveGenaralcallEvent*** Received the general call address after a start or repeated start.
***kI2C_SlaveAllEvents*** A bit mask of all available events.

#### 12.2.6.8 anonymous enum

Enumerator

***kClearFlags*** All flags which are cleared by the driver upon starting a transfer.

### 12.2.7 Function Documentation

#### 12.2.7.1 void I2C_MasterInit ( I2C_Type ∗ *base*, const i2c_master_config_t ∗ *masterConfig*, uint32_t *srcClock_Hz* )

Call this API to ungate the I2C clock and configure the I2C with master configuration.

Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can be custom filled or it can be set with default values by using the I2C_MasterGetDefaultConfig(). After calling this API, the master is ready to transfer. This is an example.

```
* i2c_master_config_t config = {
* .enableMaster = true,
* .enableStopHold = false,
* .highDrive = false,
* .baudRate_Bps = 100000,
* .glitchFilterWidth = 0
* };
* I2C_MasterInit(I2C0, &config, 12000000U);
*
```

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *masterConfig* | A pointer to the master configuration structure |
| *srcClock_Hz* | I2C peripheral clock frequency in Hz |

#### 12.2.7.2 void I2C_SlaveInit ( I2C_Type ∗ *base*, const i2c_slave_config_t ∗ *slaveConfig*, uint32_t *srcClock_Hz* )

Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can partly be set with default values by I2C_SlaveGetDefaultConfig() or it can be custom filled by the user. This is an example.

```
* i2c_slave_config_t config = {
* .enableSlave = true,
* .enableGeneralCall = false,
* .addressingMode = kI2C_Address7bit,
* .slaveAddress = 0x1DU,
* .enableWakeUp = false,
* .enablehighDrive = false,
* .enableBaudRateCtl = false,
* .sclStopHoldTime_ns = 4000
* };
* I2C_SlaveInit(I2C0, &config, 12000000U);
*
```

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *slaveConfig* | A pointer to the slave configuration structure |
| *srcClock_Hz* | I2C peripheral clock frequency in Hz |

#### 12.2.7.3 void I2C_MasterDeinit ( I2C_Type ∗ *base* )

Call this API to gate the I2C clock. The I2C master module can't work unless the I2C_MasterInit is called.

Parameters

| | |
|---|---|
| *base* | I2C base pointer |

### 12.2.7.4 void I2C_SlaveDeinit ( I2C_Type ∗ *base* )

Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C_SlaveInit is called to enable the clock.

Parameters

| | |
|---|---|
| *base* | I2C base pointer |

### 12.2.7.5 uint32_t I2C_GetInstance ( I2C_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | I2C peripheral base address. |

### 12.2.7.6 void I2C_MasterGetDefaultConfig ( i2c_master_config_t ∗ *masterConfig* )

The purpose of this API is to get the configuration structure initialized for use in the I2C_MasterConfigure(). Use the initialized structure unchanged in the I2C_MasterConfigure() or modify the structure before calling the I2C_MasterConfigure(). This is an example.

```
* i2c_master_config_t config;
* I2C_MasterGetDefaultConfig(&config);
*
```

Parameters

| | |
|---|---|
| *masterConfig* | A pointer to the master configuration structure. |

### 12.2.7.7 void I2C_SlaveGetDefaultConfig ( i2c_slave_config_t ∗ *slaveConfig* )

The purpose of this API is to get the configuration structure initialized for use in the I2C_SlaveConfigure(). Modify fields of the structure before calling the I2C_SlaveConfigure(). This is an example.

```
* i2c_slave_config_t config;
* I2C_SlaveGetDefaultConfig(&config);
*
```

Parameters

| | |
|---|---|
| *slaveConfig* | A pointer to the slave configuration structure. |

#### 12.2.7.8 static void I2C_Enable ( I2C_Type * *base*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *enable* | Pass true to enable and false to disable the module. |

#### 12.2.7.9 uint32_t I2C_MasterGetStatusFlags ( I2C_Type * *base* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer |

Returns

status flag, use status flag to AND _i2c_flags to get the related status.

#### 12.2.7.10 static uint32_t I2C_SlaveGetStatusFlags ( I2C_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | I2C base pointer |

Returns

status flag, use status flag to AND _i2c_flags to get the related status.

#### 12.2.7.11 static void I2C_MasterClearStatusFlags ( I2C_Type * *base*, uint32_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag.

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *statusMask* | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values:<br>• kI2C_StartDetectFlag (if available)<br>• kI2C_StopDetectFlag (if available)<br>• kI2C_ArbitrationLostFlag<br>• kI2C_IntPendingFlagFlag |

#### 12.2.7.12 static void I2C_SlaveClearStatusFlags ( I2C_Type ∗ *base*, uint32_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *statusMask* | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values:<br>• kI2C_StartDetectFlag (if available)<br>• kI2C_StopDetectFlag (if available)<br>• kI2C_ArbitrationLostFlag<br>• kI2C_IntPendingFlagFlag |

#### 12.2.7.13 void I2C_EnableInterrupts ( I2C_Type ∗ *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *mask* | interrupt source The parameter can be combination of the following source if defined:<br>• kI2C_GlobalInterruptEnable<br>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable<br>• kI2C_SdaTimeoutInterruptEnable |

#### 12.2.7.14 void I2C_DisableInterrupts ( I2C_Type ∗ *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *mask* | interrupt source The parameter can be combination of the following source if defined:<br>• kI2C_GlobalInterruptEnable<br>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable<br>• kI2C_SdaTimeoutInterruptEnable |

#### 12.2.7.15 static uint32_t I2C_GetDataRegAddr ( I2C_Type * *base* ) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

| | |
|---|---|
| *base* | I2C base pointer |

Returns

data register address

#### 12.2.7.16 void I2C_MasterSetBaudRate ( I2C_Type * *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer |
| *baudRate_Bps* | the baud rate value in bps |
| *srcClock_Hz* | Source clock |

#### 12.2.7.17 status_t I2C_MasterStart ( I2C_Type * *base,* uint8_t *address,* i2c_direction_t *direction* )

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. |

#### 12.2.7.18 status_t I2C_MasterStop ( I2C_Type ∗ *base* )

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully send the stop signal. |
| *kStatus_I2C_Timeout* | Send stop signal failed, timeout. |

#### 12.2.7.19 status_t I2C_MasterRepeatedStart ( I2C_Type ∗ *base,* uint8_t *address,* i2c_direction_t *direction* )

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. |

#### 12.2.7.20 status_t I2C_MasterWriteBlocking ( I2C_Type ∗ *base,* const uint8_t ∗ *txBuff,* size_t *txSize,* uint32_t *flags* )

Parameters

| | |
|---|---|
| *base* | The I2C peripheral base pointer. |
| *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 decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully complete the data transmission. |
| *kStatus_I2C_Arbitration-Lost* | Transfer error, arbitration lost. |
| *kStataus_I2C_Nak* | Transfer error, receive NAK during transfer. |

#### 12.2.7.21 status_t I2C_MasterReadBlocking ( I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*, uint32_t *flags* )

Note

The I2C_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

Parameters

| | |
|---|---|
| *base* | I2C peripheral base pointer. |
| *rxBuff* | The pointer to the data to store the received data. |
| *rxSize* | The length in bytes of the data to be received. |
| *flags* | Transfer control flag to decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully complete the data transmission. |

| | |
|---|---|
| *kStatus_I2C_Timeout* | Send stop signal failed, timeout. |

#### 12.2.7.22 status_t I2C_SlaveWriteBlocking ( I2C_Type ∗ *base,* const uint8_t ∗ *txBuff,* size_t *txSize* )

Parameters

| | |
|---|---|
| *base* | The I2C peripheral base pointer. |
| *txBuff* | The pointer to the data to be transferred. |
| *txSize* | The length in bytes of the data to be transferred. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully complete the data transmission. |
| *kStatus_I2C_Arbitration-Lost* | Transfer error, arbitration lost. |
| *kStataus_I2C_Nak* | Transfer error, receive NAK during transfer. |

#### 12.2.7.23 status_t I2C_SlaveReadBlocking ( I2C_Type ∗ *base,* uint8_t ∗ *rxBuff,* size_t *rxSize* )

Parameters

| | |
|---|---|
| *base* | I2C peripheral base pointer. |
| *rxBuff* | The pointer to the data to store the received data. |
| *rxSize* | The length in bytes of the data to be received. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully complete data receive. |
| *kStatus_I2C_Timeout* | Wait status flag timeout. |

#### 12.2.7.24 status_t I2C_MasterTransferBlocking ( I2C_Type ∗ *base,* i2c_master_transfer_t ∗ *xfer* )

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_Arbitration-Lost* | Transfer error, arbitration lost. |
| *kStataus_I2C_Nak* | Transfer error, receive NAK during transfer. |

#### 12.2.7.25 void I2C_MasterTransferCreateHandle ( I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_callback_t *callback*, void * *userData* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_master_handle_t structure to store the transfer state. |
| *callback* | pointer to user callback function. |
| *userData* | user parameter passed to the callback function. |

#### 12.2.7.26 status_t I2C_MasterTransferNonBlocking ( I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_t * *xfer* )

Note

Calling the API returns immediately after transfer initiates. The user needs to call I2C_MasterGet-TransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_master_handle_t structure which stores the transfer state. |
| *xfer* | pointer to i2c_master_transfer_t structure. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully start the data transmission. |
| *kStatus_I2C_Busy* | Previous transmission still not finished. |
| *kStatus_I2C_Timeout* | Transfer error, wait signal timeout. |

#### 12.2.7.27 status_t I2C_MasterTransferGetCount ( I2C_Type ∗ *base*, i2c_master_handle_t ∗ *handle*, size_t ∗ *count* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_master_handle_t structure which stores the transfer state. |
| *count* | Number of bytes transferred so far by the non-blocking transaction. |

Return values

| | |
|---|---|
| *kStatus_InvalidArgument* | count is Invalid. |
| *kStatus_Success* | Successfully return the count. |

#### 12.2.7.28 status_t I2C_MasterTransferAbort ( I2C_Type ∗ *base*, i2c_master_handle_t ∗ *handle* )

Note

This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_master_handle_t structure which stores the transfer state |

Return values

| | |
|---|---|
| *kStatus_I2C_Timeout* | Timeout during polling flag. |
| *kStatus_Success* | Successfully abort the transfer. |

#### 12.2.7.29 void I2C_MasterTransferHandleIRQ ( I2C_Type * *base*, void * *i2cHandle* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *i2cHandle* | pointer to i2c_master_handle_t structure. |

#### 12.2.7.30 void I2C_SlaveTransferCreateHandle ( I2C_Type * *base*, i2c_slave_handle_t * *handle*, i2c_slave_transfer_callback_t *callback*, void * *userData* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_slave_handle_t structure to store the transfer state. |
| *callback* | pointer to user callback function. |
| *userData* | user parameter passed to the callback function. |

#### 12.2.7.31 status_t I2C_SlaveTransferNonBlocking ( I2C_Type * *base*, i2c_slave_handle_t * *handle*, uint32_t *eventMask* )

Call this API after calling the I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The k-I2C_SlaveTransmitEvent and kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, 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. |

#### 12.2.7.32 void I2C_SlaveTransferAbort ( I2C_Type * *base,* i2c_slave_handle_t * *handle* )

Note

This API can be called at any time to stop slave for handling the bus events.

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *handle* | pointer to i2c_slave_handle_t structure which stores the transfer state. |

#### 12.2.7.33 status_t I2C_SlaveTransferGetCount ( I2C_Type * *base,* i2c_slave_handle_t * *handle,* size_t * *count* )

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. |

#### 12.2.7.34 void I2C_SlaveTransferHandleIRQ ( I2C_Type * *base*, void * *i2cHandle* )

Parameters

| | |
|---|---|
| *base* | I2C base pointer. |
| *i2cHandle* | pointer to i2c_slave_handle_t structure which stores the transfer state |

## 12.3 I2C CMSIS Driver

This section describes the programming interface of the I2C Cortex Microcontroller Software Interface Standard (CMSIS) driver. This driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage methord see http://www.keil.com/pack/doc/cmsis/Driver/html/index.html.

The I2C CMSIS driver includes transactional APIs.

Transactional APIs are transaction target high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code accessing the hardware registers.

### 12.3.1 I2C CMSIS Driver

#### 12.3.1.1 Master Operation in interrupt transactional method

```
void I2C_MasterSignalEvent_t(uint32_t event)
{
    if (event == ARM_I2C_EVENT_TRANSFER_DONE)
    {
        g_MasterCompletionFlag = true;
    }
}
/*Init I2C0*/
Driver_I2C0.Initialize(I2C_MasterSignalEvent_t);

Driver_I2C0.PowerControl(ARM_POWER_FULL);

/*config transmit speed*/
Driver_I2C0.Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_STANDARD);

/*start transmit*/
Driver_I2C0.MasterTransmit(I2C_MASTER_SLAVE_ADDR, g_master_buff, I2C_DATA_LENGTH, false);

/*  Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

#### 12.3.1.2 Master Operation in DMA transactional method

```
    void I2C_MasterSignalEvent_t(uint32_t event)
    {
        /*  Transfer done */
        if (event == ARM_I2C_EVENT_TRANSFER_DONE)
        {
            g_MasterCompletionFlag = true;
        }
    }

/* Init DMAMUX and DMA/EDMA. */
    DMAMUX_Init(EXAMPLE_I2C_DMAMUX_BASEADDR)
```

```
#if defined(FSL_FEATURE_SOC_DMA_COUNT) && FSL_FEATURE_SOC_DMA_COUNT > 0U
    DMA_Init(EXAMPLE_I2C_DMA_BASEADDR);
#endif /* FSL_FEATURE_SOC_DMA_COUNT */

#if defined(FSL_FEATURE_SOC_EDMA_COUNT) && FSL_FEATURE_SOC_EDMA_COUNT > 0U
    edma_config_t edmaConfig;

    EDMA_GetDefaultConfig(&edmaConfig);
    EDMA_Init(EXAMPLE_I2C_DMA_BASEADDR, &edmaConfig);
#endif /* FSL_FEATURE_SOC_EDMA_COUNT */

    /*Init I2C0*/
    Driver_I2C0.Initialize(I2C_MasterSignalEvent_t);

    Driver_I2C0.PowerControl(ARM_POWER_FULL);

    /*config transmit speed*/
    Driver_I2C0.Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_STANDARD);

    /*start transfer*/
    Driver_I2C0.MasterReceive(I2C_MASTER_SLAVE_ADDR, g_master_buff, I2C_DATA_LENGTH, false);

    /*  Wait for transfer completed. */
    while (!g_MasterCompletionFlag)
    {
    }
    g_MasterCompletionFlag = false;
```

### 12.3.1.3 Slave Operation in interrupt transactional method

```
void I2C_SlaveSignalEvent_t(uint32_t event)
{
    /*  Transfer done */
    if (event == ARM_I2C_EVENT_TRANSFER_DONE)
    {
        g_SlaveCompletionFlag = true;
    }
}

/*Init I2C1*/
Driver_I2C1.Initialize(I2C_SlaveSignalEvent_t);

Driver_I2C1.PowerControl(ARM_POWER_FULL);

/*config slave addr*/
Driver_I2C1.Control(ARM_I2C_OWN_ADDRESS, I2C_MASTER_SLAVE_ADDR);

/*start transfer*/
Driver_I2C1.SlaveReceive(g_slave_buff, I2C_DATA_LENGTH);

/*  Wait for transfer completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;
```

## 12.4 IRQ: external interrupt (IRQ) module

The MCUXpresso SDK provides a peripheral driver for the external interrupt (IRQ) module of MCUXpresso SDK devices.

### 12.4.1 IRQ Operations

#### 12.4.1.1 IRQ Initialization Operation

The IRQ Initialize is to initialize for common configure: gate the IRQ clock, configure enabled IRQ pins for pullup, edge select and detect mode, then enable the IRQ module. The IRQ Deinitialize is used to ungate the clock.

#### 12.4.1.2 IRQ Basic Operation

The IRQ provides the function to enable/disable interrupts. IRQ still provides functions to get and clear IRQF flags.

### 12.4.2 Typical use case

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/irq

# Chapter 13
# KBI: Keyboard interrupt Driver

## 13.1 Overview

The MCUXpresso SDK provides a peripheral driver for the keyboard interrupt block of MCUXpresso SDK devices.

## 13.2 KBI Operations

### 13.2.1 KBI Initialization Operation

The KBI Initialize is to initialize for common configure: gate the KBI clock, configure enabled KBI pins, and enable the interrupt. The KBI Deinitialize is to disable the interrupt/pins and ungate the clock.

### 13.2.2 KBI Basic Operation

The KBI provide the function to enable/disable interrupts. KBI still provide functions to get and clear status flags.

## 13.3 Typical use case

## Data Structures

- struct kbi_config_t
  *KBI configuration. More...*

## Enumerations

- enum kbi_detect_mode_t {
  kKBI_EdgesDetect = 0,
  kKBI_EdgesLevelDetect }
  *KBI detection mode.*

## Driver version

- #define FSL_KBI_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))
  *KBI driver version.*

## Initialization and De-initialization

- void KBI_Init (KBI_Type ∗base, kbi_config_t ∗configure)
  *KBI initialize.*
- void KBI_Deinit (KBI_Type ∗base)
  *Deinitializes the KBI module and gates the clock.*

## KBI Basic Operation

- static void KBI_EnableInterrupts (KBI_Type *base)
  *Enables the interrupt.*
- static void KBI_DisableInterrupts (KBI_Type *base)
  *Disables the interrupt.*
- static bool KBI_IsInterruptRequestDetected (KBI_Type *base)
  *Gets the KBI interrupt event status.*
- static void KBI_ClearInterruptFlag (KBI_Type *base)
  *Clears KBI status flag.*

## 13.4 Data Structure Documentation

### 13.4.1 struct kbi_config_t

## Data Fields

- uint32_t pinsEnabled
  *The eight kbi pins, set 1 to enable the corresponding KBI interrupt pins.*
- uint32_t pinsEdge
  *The edge selection for each kbi pin: 1 – rinsing edge, 0 – falling edge.*
- kbi_detect_mode_t mode
  *The kbi detection mode.*

#### Field Documentation

**(1) uint32_t kbi_config_t::pinsEnabled**

**(2) uint32_t kbi_config_t::pinsEdge**

**(3) kbi_detect_mode_t kbi_config_t::mode**

## 13.5 Macro Definition Documentation

### 13.5.1 #define FSL_KBI_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

## 13.6 Enumeration Type Documentation

### 13.6.1 enum kbi_detect_mode_t

Enumerator

***kKBI_EdgesDetect*** The keyboard detects edges only.
***kKBI_EdgesLevelDetect*** The keyboard detects both edges and levels.

## 13.7 Function Documentation

### 13.7.1 void KBI_Init ( KBI_Type * *base*, kbi_config_t * *configure* )

This function ungates the KBI clock and initializes KBI. This function must be called before calling any other KBI driver functions.

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |
| *configure* | The KBI configuration structure pointer. |

### 13.7.2 void KBI_Deinit ( KBI_Type * *base* )

This function gates the KBI clock. As a result, the KBI module doesn't work after calling this function.

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |

### 13.7.3 static void KBI_EnableInterrupts ( KBI_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |

### 13.7.4 static void KBI_DisableInterrupts ( KBI_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |

### 13.7.5 static bool KBI_IsInterruptRequestDetected ( KBI_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |

Returns

The status of the KBI interrupt request is detected.

### 13.7.6 static void KBI_ClearInterruptFlag ( KBI_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | KBI peripheral base address. |

# Chapter 14
# PIT: Periodic Interrupt Timer

## 14.1 Overview

The MCUXpresso SDK provides a driver for the Periodic Interrupt Timer (PIT) of MCUXpresso SDK devices.

## 14.2 Function groups

The PIT driver supports operating the module as a time counter.

### 14.2.1 Initialization and deinitialization

The function PIT_Init() initializes the PIT with specified configurations. The function PIT_GetDefaultConfig() gets the default configurations. The initialization function configures the PIT operation in debug mode.

The function PIT_SetTimerChainMode() configures the chain mode operation of each PIT channel.

The function PIT_Deinit() disables the PIT timers and disables the module clock.

### 14.2.2 Timer period Operations

The function PITR_SetTimerPeriod() sets the timer period in units of count. Timers begin counting down from the value set by this function until it reaches 0.

The function PIT_GetCurrentTimerCount() reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. Users can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds.

### 14.2.3 Start and Stop timer operations

The function PIT_StartTimer() starts the timer counting. After calling this function, the timer loads the period value set earlier via the PIT_SetPeriod() function and starts counting down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function PIT_StopTimer() stops the timer counting.

### 14.2.4 Status

Provides functions to get and clear the PIT status.

### 14.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

## 14.3 Typical use case

### 14.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/pit

## Data Structures

- struct pit_config_t
  *PIT configuration structure. More...*

## Enumerations

- enum pit_chnl_t {
  kPIT_Chnl_0 = 0U,
  kPIT_Chnl_1,
  kPIT_Chnl_2,
  kPIT_Chnl_3 }
  *List of PIT channels.*
- enum pit_interrupt_enable_t { kPIT_TimerInterruptEnable = PIT_TCTRL_TIE_MASK }
  *List of PIT interrupts.*
- enum pit_status_flags_t { kPIT_TimerFlag = PIT_TFLG_TIF_MASK }
  *List of PIT status flags.*

## Driver version

- #define FSL_PIT_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))
  *PIT Driver Version 2.0.4.*

## Initialization and deinitialization

- void PIT_Init (PIT_Type ∗base, const pit_config_t ∗config)
  *Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.*
- void PIT_Deinit (PIT_Type ∗base)
  *Gates the PIT clock and disables the PIT module.*
- static void PIT_GetDefaultConfig (pit_config_t ∗config)
  *Fills in the PIT configuration structure with the default settings.*
- static void PIT_SetTimerChainMode (PIT_Type ∗base, pit_chnl_t channel, bool enable)
  *Enables or disables chaining a timer with the previous timer.*

## Interrupt Interface

- static void PIT_EnableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)
  *Enables the selected PIT interrupts.*
- static void PIT_DisableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)
  *Disables the selected PIT interrupts.*
- static uint32_t PIT_GetEnabledInterrupts (PIT_Type *base, pit_chnl_t channel)
  *Gets the enabled PIT interrupts.*

## Status Interface

- static uint32_t PIT_GetStatusFlags (PIT_Type *base, pit_chnl_t channel)
  *Gets the PIT status flags.*
- static void PIT_ClearStatusFlags (PIT_Type *base, pit_chnl_t channel, uint32_t mask)
  *Clears the PIT status flags.*

## Read and Write the timer period

- static void PIT_SetTimerPeriod (PIT_Type *base, pit_chnl_t channel, uint32_t count)
  *Sets the timer period in units of count.*
- static uint32_t PIT_GetCurrentTimerCount (PIT_Type *base, pit_chnl_t channel)
  *Reads the current timer counting value.*

## Timer Start and Stop

- static void PIT_StartTimer (PIT_Type *base, pit_chnl_t channel)
  *Starts the timer counting.*
- static void PIT_StopTimer (PIT_Type *base, pit_chnl_t channel)
  *Stops the timer counting.*

# 14.4 Data Structure Documentation

## 14.4.1 struct pit_config_t

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

The configuration structure can be made constant so it resides in flash.

## Data Fields

- bool enableRunInDebug
  *true: Timers run in debug mode; false: Timers stop in debug mode*

# 14.5 Enumeration Type Documentation

## 14.5.1 enum pit_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

***kPIT_Chnl_0*** PIT channel number 0.
***kPIT_Chnl_1*** PIT channel number 1.
***kPIT_Chnl_2*** PIT channel number 2.
***kPIT_Chnl_3*** PIT channel number 3.

### 14.5.2 enum pit_interrupt_enable_t

Enumerator

***kPIT_TimerInterruptEnable*** Timer interrupt enable.

### 14.5.3 enum pit_status_flags_t

Enumerator

***kPIT_TimerFlag*** Timer flag.

## 14.6 Function Documentation

### 14.6.1 void PIT_Init ( PIT_Type * *base*, const pit_config_t * *config* )

Note

This API should be called at the beginning of the application using the PIT driver.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *config* | Pointer to the user's PIT config structure |

### 14.6.2 void PIT_Deinit ( PIT_Type * *base* )

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |

### 14.6.3 static void PIT_GetDefaultConfig ( pit_config_t * *config* ) [inline], [static]

The default values are as follows.

```
*     config->enableRunInDebug = false;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the configuration structure. |

### 14.6.4 static void PIT_SetTimerChainMode ( PIT_Type * *base*, pit_chnl_t *channel*, bool *enable* ) [inline], [static]

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer n-1 has counted down to 0, counter n decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number which is chained with the previous timer |
| *enable* | Enable or disable chain. true: Current timer is chained with the previous timer. false: Timer doesn't chain with other timers. |

### 14.6.5 static void PIT_EnableInterrupts ( PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration pit_-interrupt_enable_t |

### 14.6.6 static void PIT_DisableInterrupts ( PIT_Type ∗ *base*, pit_chnl_t *channel*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |
| *mask* | The interrupts to disable. This is a logical OR of members of the enumeration pit_-interrupt_enable_t |

### 14.6.7 static uint32_t PIT_GetEnabledInterrupts ( PIT_Type ∗ *base*, pit_chnl_t *channel* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |

Returns

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

### 14.6.8 static uint32_t PIT_GetStatusFlags ( PIT_Type ∗ *base*, pit_chnl_t *channel* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |

Returns

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

### 14.6.9 static void PIT_ClearStatusFlags ( PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |
| *mask* | The status flags to clear. This is a logical OR of members of the enumeration pit_status_flags_t |

### 14.6.10 static void PIT_SetTimerPeriod ( PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *count* ) [inline], [static]

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note

Users can call the utility macros provided in fsl_common.h to convert to ticks.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |

| | |
|---|---|
| *count* | Timer period in units of ticks |

### 14.6.11 static uint32_t PIT_GetCurrentTimerCount ( PIT_Type * *base*, pit_chnl_t *channel* ) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number |

Returns

Current timer counting value in ticks

### 14.6.12 static void PIT_StartTimer ( PIT_Type * *base*, pit_chnl_t *channel* ) [inline], [static]

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number. |

### 14.6.13 static void PIT_StopTimer ( PIT_Type * *base*, pit_chnl_t *channel* ) [inline], [static]

This function stops every timer counting. Timers reload their periods respectively after the next time they call the PIT_DRV_StartTimer.

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *channel* | Timer channel number. |

# Chapter 15
# PWT: Pulse Width Timer

## 15.1 Overview

The MCUXpresso SDK provides a driver for the Pulse Width Timer (PWT) of MCUXpresso SDK devices.

## 15.2 Function groups

The PWT driver supports capture or measure the pulse width mapping on its input channels. The counter of PWT has two selectable clock sources, Timer clock and alternative clock. PWT module supports programmable positive or negative pulse edges, and programmable interrupt generation upon pulse width values or counter overflow.

### 15.2.1 Initialization and deinitialization

The function PWT_Init() initializes the PWT with specified configurations. The function PWT_GetDefaultConfig() gets the default configurations. The initialization function configures the PWT for the requested register update mode for register with buffers.

The function PWT_Deinit() disables the PWT counter and turns off the module clock.

### 15.2.2 Reset

The function PWT_Reset() is built into PWT as a mechanism used to reset/restart the pulse width timer.

### 15.2.3 Status

Provides functions to get and clear the PWT status.

### 15.2.4 Interrupt

Provides functions to enable/disable PWT interrupts and get current enabled interrupts.

### 15.2.5 Start & Stop timer

The function PWT_StartTimer() starts the PWT time counter.

The function PWT_StopTimer() stops the PWT time counter.

### 15.2.6 GetInterrupt

Provides functions to generate Overflow/Pulse Width Data Ready Interrupt.

### 15.2.7 Get Timer value

The function PWT_GetCurrentTimerCount() is set to read the current counter value.

The function PWT_ReadPositivePulseWidth() is set to read the positive pulse width.

The function PWT_ReadNegativePulseWidth() is set to read the negative pulse width.

### 15.2.8 PWT Operations

#### Input capture operations

The input capture operations sets up an channel for input capture.

The function EdgeCapture can be used to measure the pulse width of a signal. A channel is used during capture with the input signal coming through a channel n. The capture edge for each channel, and any filter value to be used when processing the input signal.

## 15.3 Typical use case

### 15.3.1 PWT measure

This is an example code to measure the pulse width:

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/pwt

## Data Structures

- struct pwt_config_t
  *PWT configuration structure. More...*

## Macros

- #define FSL_PWT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  *Version 2.0.1.*

## Enumerations

- enum pwt_clock_source_t {
  kPWT_TimerClock = 0U,
  kPWT_AlternativeClock }
  *PWT clock source selection.*

- enum pwt_clock_prescale_t {
kPWT_Prescale_Divide_1 = 0U,
kPWT_Prescale_Divide_2,
kPWT_Prescale_Divide_4,
kPWT_Prescale_Divide_8,
kPWT_Prescale_Divide_16,
kPWT_Prescale_Divide_32,
kPWT_Prescale_Divide_64,
kPWT_Prescale_Divide_128 }
  *PWT prescaler factor selection for clock source.*
- enum pwt_input_edge_t {
kPWT_StartFall_CaptureFall_Edge = 0U,
kPWT_StartRise_CaptureRiseAndFall_Edge,
kPWT_StartFall_CaptureRiseAndFall_Edge,
kPWT_StartRise_CaptureRise_Edge }
  *PWT Input Edge.*
- enum pwt_input_select_t {
kPWT_InputPort_0 = 0U,
kPWT_InputPort_1,
kPWT_InputPort_2,
kPWT_InputPort_3 }
  *PWT input port selection.*
- enum pwt_interrupt_enable_t {
kPWT_ModuleInterruptEnable = PWT_R1_PWTIE_MASK,
kPWT_PulseWidthReadyInterruptEnable = PWT_R1_PRDYIE_MASK,
kPWT_CounterOverflowInterruptEnable = PWT_R1_POVIE_MASK }
  *List of PWT interrupts.*
- enum pwt_status_flags_t {
kPWT_CounterOverflowFlag = PWT_R1_PWTOV_MASK,
kPWT_PulseWidthValidFlag = PWT_R1_PWTRDY_MASK }
  *List of PWT flags.*

## Functions

- static uint16_t PWT_GetCurrentTimerCount (PWT_Type ∗base)
  *Reads the current counter value.*
- static uint16_t PWT_ReadPositivePulseWidth (PWT_Type ∗base)
  *Reads the positive pulse width.*
- static uint16_t PWT_ReadNegativePulseWidth (PWT_Type ∗base)
  *Reads the negative pulse width.*
- static void PWT_Reset (PWT_Type ∗base)
  *Performs a software reset on the PWT module.*

## Initialization and deinitialization

- void PWT_Init (PWT_Type ∗base, const pwt_config_t ∗config)
  *Ungates the PWT clock and configures the peripheral for basic operation.*
- void PWT_Deinit (PWT_Type ∗base)

*Gates the PWT clock.*
- void PWT_GetDefaultConfig (pwt_config_t *config)
  *Fills in the PWT configuration structure with the default settings.*

### Interrupt Interface

- static void PWT_EnableInterrupts (PWT_Type *base, uint32_t mask)
  *Enables the selected PWT interrupts.*
- static void PWT_DisableInterrupts (PWT_Type *base, uint32_t mask)
  *Disables the selected PWT interrupts.*
- static uint32_t PWT_GetEnabledInterrupts (PWT_Type *base)
  *Gets the enabled PWT interrupts.*

### Status Interface

- static uint32_t PWT_GetStatusFlags (PWT_Type *base)
  *Gets the PWT status flags.*
- static void PWT_ClearStatusFlags (PWT_Type *base, uint32_t mask)
  *Clears the PWT status flags.*

### Timer Start and Stop

- static void PWT_StartTimer (PWT_Type *base)
  *Starts the PWT counter.*
- static void PWT_StopTimer (PWT_Type *base)
  *Stops the PWT counter.*

## 15.4 Data Structure Documentation

### 15.4.1 struct pwt_config_t

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

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

### Data Fields

- pwt_clock_source_t clockSource
  *Clock source for the counter.*
- pwt_clock_prescale_t prescale
  *Pre-scaler to divide down the clock.*
- pwt_input_select_t inputSelect
  *PWT Pulse input port selection.*
- pwt_input_edge_t edge
  *PWT Input Edge.*

## 15.5 Enumeration Type Documentation

### 15.5.1 enum pwt_clock_source_t

Enumerator

***kPWT_TimerClock*** The Timer clock is used as the clock source of PWT counter.
***kPWT_AlternativeClock*** Alternative clock is used as the clock source of PWT counter.

### 15.5.2 enum pwt_clock_prescale_t

Enumerator

***kPWT_Prescale_Divide_1*** PWT clock divided by 1.
***kPWT_Prescale_Divide_2*** PWT clock divided by 2.
***kPWT_Prescale_Divide_4*** PWT clock divided by 4.
***kPWT_Prescale_Divide_8*** PWT clock divided by 8.
***kPWT_Prescale_Divide_16*** PWT clock divided by 16.
***kPWT_Prescale_Divide_32*** PWT clock divided by 32.
***kPWT_Prescale_Divide_64*** PWT clock divided by 64.
***kPWT_Prescale_Divide_128*** PWT clock divided by 128.

### 15.5.3 enum pwt_input_edge_t

Enumerator

***kPWT_StartFall_CaptureFall_Edge*** The first falling-edge starts the pulse width measurement, and on all the subsequent falling edges, the pulse width is captured.
***kPWT_StartRise_CaptureRiseAndFall_Edge*** The first rising edge starts the pulse width measurement, and on all the subsequent rising and falling edges, the pulse width is captured.
***kPWT_StartFall_CaptureRiseAndFall_Edge*** The first falling edge starts the pulse width measurement, and on all the subsequent rising and falling edges, the pulse width is captured.
***kPWT_StartRise_CaptureRise_Edge*** The first-rising edge starts the pulse width measurement, and on all the subsequent rising edges, the pulse width is captured.

### 15.5.4 enum pwt_input_select_t

Enumerator

***kPWT_InputPort_0*** PWT input comes from PWTIN[0].
***kPWT_InputPort_1*** PWT input comes from PWTIN[1].
***kPWT_InputPort_2*** PWT input comes from PWTIN[2].
***kPWT_InputPort_3*** PWT input comes from PWTIN[3].

### 15.5.5 enum pwt_interrupt_enable_t

Enumerator

***kPWT_ModuleInterruptEnable*** Module Interrupt.
***kPWT_PulseWidthReadyInterruptEnable*** Pulse width data ready interrupt.
***kPWT_CounterOverflowInterruptEnable*** Counter overflow interrupt.

### 15.5.6 enum pwt_status_flags_t

Enumerator

***kPWT_CounterOverflowFlag*** Counter overflow flag.
***kPWT_PulseWidthValidFlag*** Pulse width valid flag.

## 15.6 Function Documentation

### 15.6.1 void PWT_Init ( PWT_Type * *base*, const pwt_config_t * *config* )

Note

This API should be called at the beginning of the application using the PWT driver.

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |
| *config* | Pointer to the user configuration structure. |

### 15.6.2 void PWT_Deinit ( PWT_Type * *base* )

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

### 15.6.3 void PWT_GetDefaultConfig ( pwt_config_t * *config* )

The default values are:

```
*   config->clockSource = kPWT_TimerClock;
*   config->prescale = kPWT_Prescale_Divide_1;
*   config->inputSelect = kPWT_InputPort_0;
*   config->edge = kPWT_StartRise_CaptureRiseAndFall_Edge;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the user configuration structure. |

### 15.6.4 static void PWT_EnableInterrupts ( PWT_Type ∗ *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration pwt_-interrupt_enable_t |

### 15.6.5 static void PWT_DisableInterrupts ( PWT_Type ∗ *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration pwt_-interrupt_enable_t |

### 15.6.6 static uint32_t PWT_GetEnabledInterrupts ( PWT_Type ∗ *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

Returns

The enabled interrupts. This is the logical OR of members of the enumeration pwt_interrupt_enable-_t

### 15.6.7 static uint32_t PWT_GetStatusFlags ( PWT_Type ∗ *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

Returns

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

### 15.6.8 static void PWT_ClearStatusFlags ( PWT_Type ∗ *base,* uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |
| *mask* | The status flags to clear. This is a logical OR of members of the enumeration pwt_-status_flags_t |

### 15.6.9 static void PWT_StartTimer ( PWT_Type ∗ *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

### 15.6.10 static void PWT_StopTimer ( PWT_Type ∗ *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

### 15.6.11 static uint16_t PWT_GetCurrentTimerCount ( PWT_Type ∗ *base* ) [inline], [static]

This function returns the timer counting value

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

Returns

Current 16-bit timer counter value

### 15.6.12 static uint16_t PWT_ReadPositivePulseWidth ( PWT_Type * *base* ) [inline], [static]

This function reads the low and high registers and returns the 16-bit positive pulse width

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address. |

Returns

The 16-bit positive pulse width.

### 15.6.13 static uint16_t PWT_ReadNegativePulseWidth ( PWT_Type * *base* ) [inline], [static]

This function reads the low and high registers and returns the 16-bit negative pulse width

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address. |

Returns

The 16-bit negative pulse width.

### 15.6.14 static void PWT_Reset ( PWT_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PWT peripheral base address |

# Chapter 16
# RTC: Real Time Clock

## 16.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Real Time Clock module of MCUXpresso SDK devices.

## 16.2 Typical use case

Example use of RTC API. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/rtc/

### Data Structures

- struct rtc_datetime_t
  *Structure is used to hold the date and time. More...*
- struct rtc_config_t
  *RTC config structure. More...*

### Typedefs

- typedef void(∗ rtc_alarm_callback_t )(void)
  *RTC alarm callback function.*

### Enumerations

- enum rtc_clock_source_t {
  kRTC_ExternalClock = 0U,
  kRTC_LPOCLK = 1U,
  kRTC_ICSIRCLK = 2U,
  kRTC_BusClock = 3U }
  *List of RTC clock source.*
- enum rtc_clock_prescaler_t {
  kRTC_ClockDivide_off = 0U,
  kRTC_ClockDivide_1_128 = 1U,
  kRTC_ClockDivide_2_256 = 2U,
  kRTC_ClockDivide_4_512 = 3U,
  kRTC_ClockDivide_8_1024 = 4U,
  kRTC_ClockDivide_16_2048 = 5U,
  kRTC_ClockDivide_32_100 = 6U,
  kRTC_ClockDivide_64_1000 = 7U }
  *List of RTC clock prescaler.*
- enum rtc_interrupt_enable_t { kRTC_InterruptEnable = RTC_SC_RTIE_MASK }

*List of RTC interrupts.*
- enum rtc_interrupt_flags_t { kRTC_InterruptFlag = RTC_SC_RTIF_MASK }
  *List of RTC Interrupt flags.*
- enum rtc_output_enable_t { kRTC_OutputEnable = RTC_SC_RTCO_MASK }
  *List of RTC Output.*

## Driver version

- #define FSL_RTC_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))
  *Version 2.0.4.*

## Initialization and deinitialization

- void RTC_Init (RTC_Type ∗base, const rtc_config_t ∗config)
  *Ungates the RTC clock and configures the peripheral for basic operation.*
- void RTC_Deinit (RTC_Type ∗base)
  *Stops the timer and gate the RTC clock.*
- void RTC_GetDefaultConfig (rtc_config_t ∗config)
  *Fills in the RTC config struct with the default settings.*

## Current Time & Alarm

- status_t RTC_SetDatetime (rtc_datetime_t ∗datetime)
  *Sets the RTC date and time according to the given time structure.*
- void RTC_GetDatetime (rtc_datetime_t ∗datetime)
  *Gets the RTC time and stores it in the given time structure.*
- void RTC_SetAlarm (uint32_t second)
  *Sets the RTC alarm time.*
- void RTC_GetAlarm (rtc_datetime_t ∗datetime)
  *Returns the RTC alarm time.*
- void RTC_SetAlarmCallback (rtc_alarm_callback_t callback)
  *Set the RTC alarm callback.*

## Select Source clock

- static void RTC_SelectSourceClock (RTC_Type ∗base, rtc_clock_source_t clock, rtc_clock_prescaler_t divide)
  *Select Real-Time Clock Source and Clock Prescaler.*
- uint32_t RTC_GetDivideValue (RTC_Type ∗base)
  *Get the RTC Divide value.*

## Interrupt Interface

- static void RTC_EnableInterrupts (RTC_Type ∗base, uint32_t mask)
  *Enables the selected RTC interrupts.*
- static void RTC_DisableInterrupts (RTC_Type ∗base, uint32_t mask)
  *Disables the selected RTC interrupts.*
- static uint32_t RTC_GetEnabledInterrupts (RTC_Type ∗base)
  *Gets the enabled RTC interrupts.*
- static uint32_t RTC_GetInterruptFlags (RTC_Type ∗base)

*Gets the RTC interrupt flags.*
- static void RTC_ClearInterruptFlags (RTC_Type ∗base, uint32_t mask)
  *Clears the RTC interrupt flags.*

## Output Interface

- static void RTC_EnableOutput (RTC_Type ∗base, uint32_t mask)
  *Enable the RTC output.*
- static void RTC_DisableOutput (RTC_Type ∗base, uint32_t mask)
  *Disable the RTC output.*

## Set module value and Get Count value

- static void RTC_SetModuloValue (RTC_Type ∗base, uint32_t value)
  *Set the RTC module value.*
- static uint16_t RTC_GetCountValue (RTC_Type ∗base)
  *Get the RTC Count value.*

# 16.3 Data Structure Documentation

## 16.3.1 struct rtc_datetime_t

## Data Fields

- 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.*

### Field Documentation

#### (1) uint16_t rtc_datetime_t::year

#### (2) uint8_t rtc_datetime_t::month

#### (3) uint8_t rtc_datetime_t::day

#### (4) uint8_t rtc_datetime_t::hour

#### (5) uint8_t rtc_datetime_t::minute

#### (6) uint8_t rtc_datetime_t::second

### 16.3.2 struct rtc_config_t

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

## 16.4 Typedef Documentation

### 16.4.1 typedef void(∗ rtc_alarm_callback_t)(void)

## 16.5 Enumeration Type Documentation

### 16.5.1 enum rtc_clock_source_t

Enumerator

***kRTC_ExternalClock*** External clock source.
***kRTC_LPOCLK*** Real-time clock source is 1 kHz (LPOCLK)
***kRTC_ICSIRCLK*** Internal reference clock (ICSIRCLK)
***kRTC_BusClock*** Bus clock.

### 16.5.2 enum rtc_clock_prescaler_t

Enumerator

***kRTC_ClockDivide_off*** Off.
***kRTC_ClockDivide_1_128*** If RTCLKS = x0, it is 1; if RTCLKS = x1, it is 128.
***kRTC_ClockDivide_2_256*** If RTCLKS = x0, it is 2; if RTCLKS = x1, it is 256.
***kRTC_ClockDivide_4_512*** If RTCLKS = x0, it is 4; if RTCLKS = x1, it is 512.
***kRTC_ClockDivide_8_1024*** If RTCLKS = x0, it is 8; if RTCLKS = x1, it is 1024.
***kRTC_ClockDivide_16_2048*** If RTCLKS = x0, it is 16; if RTCLKS = x1, it is 2048.
***kRTC_ClockDivide_32_100*** If RTCLKS = x0, it is 32; if RTCLKS = x1, it is 100.
***kRTC_ClockDivide_64_1000*** If RTCLKS = x0, it is 64; if RTCLKS = x1, it is 1000.

### 16.5.3 enum rtc_interrupt_enable_t

Enumerator

***kRTC_InterruptEnable*** Interrupt enable.

### 16.5.4 enum rtc_interrupt_flags_t

Enumerator

***kRTC_InterruptFlag*** Interrupt flag.

### 16.5.5 enum rtc_output_enable_t

Enumerator

***kRTC_OutputEnable*** Output enable.

## 16.6 Function Documentation

### 16.6.1 void RTC_Init ( RTC_Type * *base*, const rtc_config_t * *config* )

Note

This API should be called at the beginning of the application using the RTC driver.

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *config* | Pointer to the user's RTC configuration structure. |

### 16.6.2 void RTC_Deinit ( RTC_Type * *base* )

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |

### 16.6.3 void RTC_GetDefaultConfig ( rtc_config_t * *config* )

The default values are as follows.

```
*    config->clockSource = kRTC_BusClock;
*    config->prescaler = kRTC_ClockDivide_16_2048;
*    config->time_us = 1000000U;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the user's RTC configuration structure. |

### 16.6.4 status_t RTC_SetDatetime ( rtc_datetime_t * *datetime* )

Parameters

| | |
|---|---|
| *datetime* | Pointer to the structure where the date and time details are stored. |

Returns

kStatus_Success: Success in setting the time and starting the RTC kStatus_InvalidArgument: Error because the datetime format is incorrect

### 16.6.5 void RTC_GetDatetime ( rtc_datetime_t * *datetime* )

Parameters

| | |
|---|---|
| *datetime* | Pointer to the structure where the date and time details are stored. |

### 16.6.6 void RTC_SetAlarm ( uint32_t *second* )

Parameters

| | |
|---|---|
| *second* | Second value. User input the number of second. After seconds user input, alarm occurs. |

### 16.6.7 void RTC_GetAlarm ( rtc_datetime_t * *datetime* )

Parameters

| | |
|---|---|
| *datetime* | Pointer to the structure where the alarm date and time details are stored. |

### 16.6.8 void RTC_SetAlarmCallback ( rtc_alarm_callback_t *callback* )

Parameters

| | |
|---|---|
| *callback* | The callback function. |

### 16.6.9 static void RTC_SelectSourceClock ( RTC_Type ∗ *base*, rtc_clock_source_t *clock*, rtc_clock_prescaler_t *divide* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *clock* | Select RTC clock source |
| *divide* | Select RTC clock prescaler value |

### 16.6.10 uint32_t RTC_GetDivideValue ( RTC_Type ∗ *base* )

Note

This API should be called after selecting clock source and clock prescaler.

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |

Returns

The Divider value. The Divider value depends on clock source and clock prescaler

### 16.6.11 static void RTC_EnableInterrupts ( RTC_Type ∗ *base*, uint32_t *mask* ) [inline], [static]

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 |

### 16.6.12 static void RTC_DisableInterrupts ( RTC_Type * *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | PIT peripheral base address |
| *mask* | The interrupts to disable. This is a logical OR of members of the enumeration rtc_-interrupt_enable_t |

### 16.6.13 static uint32_t RTC_GetEnabledInterrupts ( RTC_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |

Returns

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

### 16.6.14 static uint32_t RTC_GetInterruptFlags ( RTC_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |

Returns

The interrupt flags. This is the logical OR of members of the enumeration rtc_interrupt_flags_t

### 16.6.15 static void RTC_ClearInterruptFlags ( RTC_Type * *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *mask* | The interrupt flags to clear. This is a logical OR of members of the enumeration rtc_interrupt_flags_t |

### 16.6.16 static void RTC_EnableOutput ( RTC_Type * *base*, uint32_t *mask* ) [inline], [static]

If RTC output is enabled, the RTCO pinout will be toggled when RTC counter overflows

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *mask* | The Output to enable. This is a logical OR of members of the enumeration rtc_output-_enable_t |

### 16.6.17 static void RTC_DisableOutput ( RTC_Type * *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *mask* | The Output to disable. This is a logical OR of members of the enumeration rtc_-output_enable_t |

### 16.6.18 static void RTC_SetModuloValue ( RTC_Type * *base*, uint32_t *value* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |
| *value* | The Module Value. The RTC Modulo register allows the compare value to be set to any value from 0x0000 to 0xFFFF |

### 16.6.19 static uint16_t RTC_GetCountValue ( RTC_Type * base ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | RTC peripheral base address |

Returns

The Count Value. The Count Value is allowed from 0x0000 to 0xFFFF

# Chapter 17
# SPI: Serial Peripheral Interface Driver

## 17.1 Overview

### Modules

- SPI CMSIS driver
- SPI Driver

## 17.2 SPI Driver

### 17.2.1 Overview

SPI driver includes functional APIs and transactional APIs.

Functional APIs are feature/property target low level APIs. Functional APIs can be used for SPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the SPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. SPI functional operation groups provide the functional API set.

Transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional A-PI implementation and write a custom code. All transactional APIs use the spi_handle_t as the first parameter. Initialize the handle by calling the SPI_MasterTransferCreateHandle() or SPI_SlaveTransfer-CreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions SPI_MasterTransferNon-Blocking() and SPI_SlaveTransferNonBlocking() set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_SPI_Idle status.

### 17.2.2 Typical use case

#### 17.2.2.1 SPI master transfer using an interrupt method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/spi

#### 17.2.2.2 SPI Send/receive using a DMA method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/spi

### Data Structures

- struct spi_master_config_t
  *SPI master user configure structure.* More...
- struct spi_slave_config_t
  *SPI slave user configure structure.* More...
- struct spi_transfer_t
  *SPI transfer structure.* More...
- struct spi_master_handle_t
  *SPI transfer handle structure.* More...

## Macros

- #define SPI_DUMMYDATA (0xFFU)
  *SPI dummy transfer data, the data is sent while txBuff is NULL.*
- #define SPI_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/
  *Retry times for waiting flag.*

## Typedefs

- typedef spi_master_handle_t spi_slave_handle_t
  *Slave handle is the same with master handle.*
- 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 master callback for finished transmit.*

## Enumerations

- enum {
  kStatus_SPI_Busy = MAKE_STATUS(kStatusGroup_SPI, 0),
  kStatus_SPI_Idle = MAKE_STATUS(kStatusGroup_SPI, 1),
  kStatus_SPI_Error = MAKE_STATUS(kStatusGroup_SPI, 2),
  kStatus_SPI_Timeout = MAKE_STATUS(kStatusGroup_SPI, 3) }
  *Return status for the SPI driver.*
- enum spi_clock_polarity_t {
  kSPI_ClockPolarityActiveHigh = 0x0U,
  kSPI_ClockPolarityActiveLow }
  *SPI clock polarity configuration.*
- enum spi_clock_phase_t {
  kSPI_ClockPhaseFirstEdge = 0x0U,
  kSPI_ClockPhaseSecondEdge }
  *SPI clock phase configuration.*
- enum spi_shift_direction_t {
  kSPI_MsbFirst = 0x0U,
  kSPI_LsbFirst }
  *SPI data shifter direction options.*
- enum spi_ss_output_mode_t {
  kSPI_SlaveSelectAsGpio = 0x0U,
  kSPI_SlaveSelectFaultInput = 0x2U,
  kSPI_SlaveSelectAutomaticOutput = 0x3U }
  *SPI slave select output mode options.*
- enum spi_pin_mode_t {

kSPI_PinModeNormal = 0x0U,
kSPI_PinModeInput = 0x1U,
kSPI_PinModeOutput = 0x3U }
*SPI pin mode options.*
- enum spi_data_bitcount_mode_t {
kSPI_8BitMode = 0x0U,
kSPI_16BitMode }
*SPI data length mode options.*
- enum _spi_interrupt_enable {
kSPI_RxFullAndModfInterruptEnable = 0x1U,
kSPI_TxEmptyInterruptEnable = 0x2U,
kSPI_MatchInterruptEnable = 0x4U }
*SPI interrupt sources.*
- enum _spi_flags {
kSPI_RxBufferFullFlag = SPI_S_SPRF_MASK,
kSPI_MatchFlag = SPI_S_SPMF_MASK,
kSPI_TxBufferEmptyFlag = SPI_S_SPTEF_MASK,
kSPI_ModeFaultFlag = SPI_S_MODF_MASK }
*SPI status flags.*

## Variables

- volatile uint8_t g_spiDummyData [ ]
*Global variable for dummy data value setting.*

## Driver version

- #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))
*SPI driver version.*

## Initialization and deinitialization

- void SPI_MasterGetDefaultConfig (spi_master_config_t ∗config)
*Sets the SPI master configuration structure to default values.*
- void SPI_MasterInit (SPI_Type ∗base, const spi_master_config_t ∗config, uint32_t srcClock_Hz)
*Initializes the SPI with master configuration.*
- void SPI_SlaveGetDefaultConfig (spi_slave_config_t ∗config)
*Sets the SPI slave configuration structure to default values.*
- void SPI_SlaveInit (SPI_Type ∗base, const spi_slave_config_t ∗config)
*Initializes the SPI with slave configuration.*
- void SPI_Deinit (SPI_Type ∗base)
*De-initializes the SPI.*
- static void SPI_Enable (SPI_Type ∗base, bool enable)
*Enables or disables the SPI.*

### Status

- uint32_t SPI_GetStatusFlags (SPI_Type *base)
  *Gets the status flag.*

### Interrupts

- void SPI_EnableInterrupts (SPI_Type *base, uint32_t mask)
  *Enables the interrupt for the SPI.*
- void SPI_DisableInterrupts (SPI_Type *base, uint32_t mask)
  *Disables the interrupt for the SPI.*

### DMA Control

- static uint32_t SPI_GetDataRegisterAddress (SPI_Type *base)
  *Gets the SPI tx/rx data register address.*

### Bus Operations

- uint32_t SPI_GetInstance (SPI_Type *base)
  *Get the instance for SPI module.*
- static void SPI_SetPinMode (SPI_Type *base, spi_pin_mode_t pinMode)
  *Sets the pin mode for transfer.*
- void SPI_MasterSetBaudRate (SPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  *Sets the baud rate for SPI transfer.*
- static void SPI_SetMatchData (SPI_Type *base, uint32_t matchData)
  *Sets the match data for SPI.*
- status_t SPI_WriteBlocking (SPI_Type *base, uint8_t *buffer, size_t size)
  *Sends a buffer of data bytes using a blocking method.*
- void SPI_WriteData (SPI_Type *base, uint16_t data)
  *Writes a data into the SPI data register.*
- uint16_t SPI_ReadData (SPI_Type *base)
  *Gets a data from the SPI data register.*
- void SPI_SetDummyData (SPI_Type *base, uint8_t dummyData)
  *Set up the dummy data.*

### Transactional

- void SPI_MasterTransferCreateHandle (SPI_Type *base, spi_master_handle_t *handle, spi_master_callback_t callback, void *userData)
  *Initializes the SPI master handle.*
- status_t SPI_MasterTransferBlocking (SPI_Type *base, spi_transfer_t *xfer)
  *Transfers a block of data using a polling method.*
- status_t SPI_MasterTransferNonBlocking (SPI_Type *base, spi_master_handle_t *handle, spi_transfer_t *xfer)

*Performs a non-blocking SPI interrupt transfer.*
- status_t SPI_MasterTransferGetCount (SPI_Type ∗base, spi_master_handle_t ∗handle, size_t ∗count)
  *Gets the bytes of the SPI interrupt transferred.*
- void SPI_MasterTransferAbort (SPI_Type ∗base, spi_master_handle_t ∗handle)
  *Aborts an SPI transfer using interrupt.*
- void SPI_MasterTransferHandleIRQ (SPI_Type ∗base, spi_master_handle_t ∗handle)
  *Interrupts the handler for the SPI.*
- void SPI_SlaveTransferCreateHandle (SPI_Type ∗base, spi_slave_handle_t ∗handle, spi_slave_callback_t callback, void ∗userData)
  *Initializes the SPI slave handle.*
- status_t SPI_SlaveTransferNonBlocking (SPI_Type ∗base, spi_slave_handle_t ∗handle, spi_transfer_t ∗xfer)
  *Performs a non-blocking SPI slave interrupt transfer.*
- static status_t SPI_SlaveTransferGetCount (SPI_Type ∗base, spi_slave_handle_t ∗handle, size_t ∗count)
  *Gets the bytes of the SPI interrupt transferred.*
- static void SPI_SlaveTransferAbort (SPI_Type ∗base, spi_slave_handle_t ∗handle)
  *Aborts an SPI slave transfer using interrupt.*
- void SPI_SlaveTransferHandleIRQ (SPI_Type ∗base, spi_slave_handle_t ∗handle)
  *Interrupts a handler for the SPI slave.*

## 17.2.3 Data Structure Documentation

### 17.2.3.1 struct spi_master_config_t

#### Data Fields

- bool enableMaster
  *Enable SPI at initialization time.*
- bool enableStopInWaitMode
  *SPI stop in wait mode.*
- spi_clock_polarity_t polarity
  *Clock polarity.*
- spi_clock_phase_t phase
  *Clock phase.*
- spi_shift_direction_t direction
  *MSB or LSB.*
- spi_ss_output_mode_t outputMode
  *SS pin setting.*
- spi_pin_mode_t pinMode
  *SPI pin mode select.*
- uint32_t baudRate_Bps
  *Baud Rate for SPI in Hz.*

#### 17.2.3.2 struct spi_slave_config_t

**Data Fields**

- bool enableSlave
  *Enable SPI at initialization time.*
- bool enableStopInWaitMode
  *SPI stop in wait mode.*
- spi_clock_polarity_t polarity
  *Clock polarity.*
- spi_clock_phase_t phase
  *Clock phase.*
- spi_shift_direction_t direction
  *MSB or LSB.*
- spi_pin_mode_t pinMode
  *SPI pin mode select.*

#### 17.2.3.3 struct spi_transfer_t

**Data Fields**

- uint8_t * txData
  *Send buffer.*
- uint8_t * rxData
  *Receive buffer.*
- size_t dataSize
  *Transfer bytes.*
- uint32_t flags
  *SPI control flag, useless to SPI.*

**Field Documentation**

**(1) uint32_t spi_transfer_t::flags**

#### 17.2.3.4 struct _spi_master_handle

**Data Fields**

- uint8_t *volatile txData
  *Transfer buffer.*
- uint8_t *volatile rxData
  *Receive buffer.*
- volatile size_t txRemainingBytes
  *Send data remaining in bytes.*
- volatile size_t rxRemainingBytes
  *Receive data remaining in bytes.*
- volatile uint32_t state
  *SPI internal state.*
- size_t transferSize
  *Bytes to be transferred.*

- uint8_t bytePerFrame
  *SPI mode, 2bytes or 1byte in a frame.*
- uint8_t watermark
  *Watermark value for SPI transfer.*
- spi_master_callback_t callback
  *SPI callback.*
- void ∗ userData
  *Callback parameter.*

### 17.2.4 Macro Definition Documentation

#### 17.2.4.1 #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

#### 17.2.4.2 #define SPI_DUMMYDATA (0xFFU)

#### 17.2.4.3 #define SPI_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/

### 17.2.5 Enumeration Type Documentation

#### 17.2.5.1 anonymous enum

Enumerator

***kStatus_SPI_Busy*** SPI bus is busy.
***kStatus_SPI_Idle*** SPI is idle.
***kStatus_SPI_Error*** SPI error.
***kStatus_SPI_Timeout*** SPI timeout polling status flags.

#### 17.2.5.2 enum spi_clock_polarity_t

Enumerator

***kSPI_ClockPolarityActiveHigh*** Active-high SPI clock (idles low).
***kSPI_ClockPolarityActiveLow*** Active-low SPI clock (idles high).

#### 17.2.5.3 enum spi_clock_phase_t

Enumerator

***kSPI_ClockPhaseFirstEdge*** First edge on SPSCK occurs at the middle of the first cycle of a data transfer.
***kSPI_ClockPhaseSecondEdge*** First edge on SPSCK occurs at the start of the first cycle of a data transfer.

#### 17.2.5.4 enum spi_shift_direction_t

Enumerator

***kSPI_MsbFirst*** Data transfers start with most significant bit.
***kSPI_LsbFirst*** Data transfers start with least significant bit.

#### 17.2.5.5 enum spi_ss_output_mode_t

Enumerator

***kSPI_SlaveSelectAsGpio*** Slave select pin configured as GPIO.
***kSPI_SlaveSelectFaultInput*** Slave select pin configured for fault detection.
***kSPI_SlaveSelectAutomaticOutput*** Slave select pin configured for automatic SPI output.

#### 17.2.5.6 enum spi_pin_mode_t

Enumerator

***kSPI_PinModeNormal*** Pins operate in normal, single-direction mode.
***kSPI_PinModeInput*** Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input.
***kSPI_PinModeOutput*** Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output.

#### 17.2.5.7 enum spi_data_bitcount_mode_t

Enumerator

***kSPI_8BitMode*** 8-bit data transmission mode
***kSPI_16BitMode*** 16-bit data transmission mode

#### 17.2.5.8 enum _spi_interrupt_enable

Enumerator

***kSPI_RxFullAndModfInterruptEnable*** Receive buffer full (SPRF) and mode fault (MODF) interrupt.
***kSPI_TxEmptyInterruptEnable*** Transmit buffer empty interrupt.
***kSPI_MatchInterruptEnable*** Match interrupt.

#### 17.2.5.9 enum _spi_flags

Enumerator

***kSPI_RxBufferFullFlag*** Read buffer full flag.
***kSPI_MatchFlag*** Match flag.
***kSPI_TxBufferEmptyFlag*** Transmit buffer empty flag.
***kSPI_ModeFaultFlag*** Mode fault flag.

### 17.2.6 Function Documentation

#### 17.2.6.1 void SPI_MasterGetDefaultConfig ( spi_master_config_t * *config* )

The purpose of this API is to get the configuration structure initialized for use in SPI_MasterInit(). User may use the initialized structure unchanged in SPI_MasterInit(), or modify some fields of the structure before calling SPI_MasterInit(). After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

Parameters

| | |
|---|---|
| *config* | pointer to master config structure |

#### 17.2.6.2 void SPI_MasterInit ( SPI_Type * *base*, const spi_master_config_t * *config*, uint32_t *srcClock_Hz* )

The configuration structure can be filled by user from scratch, or be set with default values by SPI_Master-GetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
.baudRate_Bps = 400000,
...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

| | |
|---|---|
| *base* | SPI base pointer |

| | |
|---|---|
| *config* | pointer to master configuration structure |
| *srcClock_Hz* | Source clock frequency. |

#### 17.2.6.3 void SPI_SlaveGetDefaultConfig ( spi_slave_config_t * *config* )

The purpose of this API is to get the configuration structure initialized for use in SPI_SlaveInit(). Modify some fields of the structure before calling SPI_SlaveInit(). Example:

```
spi_slave_config_t config;
SPI_SlaveGetDefaultConfig(&config);
```

Parameters

| | |
|---|---|
| *config* | pointer to slave configuration structure |

#### 17.2.6.4 void SPI_SlaveInit ( SPI_Type * *base*, const spi_slave_config_t * *config* )

The configuration structure can be filled by user from scratch or be set with default values by SPI_Slave-GetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {
.polarity = kSPIClockPolarity_ActiveHigh;
.phase = kSPIClockPhase_FirstEdge;
.direction = kSPIMsbFirst;
...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *config* | pointer to master configuration structure |

#### 17.2.6.5 void SPI_Deinit ( SPI_Type * *base* )

Calling this API resets the SPI module, gates the SPI clock. The SPI module can't work unless calling the SPI_MasterInit/SPI_SlaveInit to initialize module.

Parameters

| | |
|---|---|
| *base* | SPI base pointer |

### 17.2.6.6 static void SPI_Enable ( SPI_Type ∗ *base*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *enable* | pass true to enable module, false to disable module |

### 17.2.6.7 uint32_t SPI_GetStatusFlags ( SPI_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |

Returns

SPI Status, use status flag to AND _spi_flags could get the related status.

### 17.2.6.8 void SPI_EnableInterrupts ( SPI_Type ∗ *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *mask* | SPI interrupt source. The parameter can be any combination of the following values:<br>• kSPI_RxFullAndModfInterruptEnable<br>• kSPI_TxEmptyInterruptEnable<br>• kSPI_MatchInterruptEnable<br>• kSPI_RxFifoNearFullInterruptEnable<br>• kSPI_TxFifoNearEmptyInterruptEnable |

### 17.2.6.9 void SPI_DisableInterrupts ( SPI_Type ∗ *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *mask* | SPI interrupt source. The parameter can be any combination of the following values:<br>• kSPI_RxFullAndModfInterruptEnable<br>• kSPI_TxEmptyInterruptEnable<br>• kSPI_MatchInterruptEnable<br>• kSPI_RxFifoNearFullInterruptEnable<br>• kSPI_TxFifoNearEmptyInterruptEnable |

#### 17.2.6.10 static uint32_t SPI_GetDataRegisterAddress ( SPI_Type ∗ *base* ) `[inline]`, `[static]`

This API is used to provide a transfer address for the SPI DMA transfer configuration.

Parameters

| | |
|---|---|
| *base* | SPI base pointer |

Returns

data register address

#### 17.2.6.11 uint32_t SPI_GetInstance ( SPI_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | SPI base address |

#### 17.2.6.12 static void SPI_SetPinMode ( SPI_Type ∗ *base*, spi_pin_mode_t *pinMode* ) `[inline]`, `[static]`

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *pinMode* | pin mode for transfer AND _spi_pin_mode could get the related configuration. |

#### 17.2.6.13 void SPI_MasterSetBaudRate ( SPI_Type ∗ *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz* )

This is only used in master.

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *baudRate_Bps* | baud rate needed in Hz. |
| *srcClock_Hz* | SPI source clock frequency in Hz. |

#### 17.2.6.14 static void SPI_SetMatchData ( SPI_Type ∗ *base*, uint32_t *matchData* ) [inline], [static]

The match data is a hardware comparison value. When the value received in the SPI receive data buffer equals the hardware comparison value, the SPI Match Flag in the S register (S[SPMF]) sets. This can also generate an interrupt if the enable bit sets.

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *matchData* | Match data. |

#### 17.2.6.15 status_t SPI_WriteBlocking ( SPI_Type ∗ *base*, uint8_t ∗ *buffer*, size_t *size* )

Note

This function blocks via polling until all bytes have been sent.

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *buffer* | The data bytes to send |
| *size* | The number of data bytes to send |

Returns

kStatus_SPI_Timeout The transfer timed out and was aborted.

#### 17.2.6.16 void SPI_WriteData ( SPI_Type * *base*, uint16_t *data* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *data* | needs to be write. |

#### 17.2.6.17 uint16_t SPI_ReadData ( SPI_Type * *base* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |

Returns

Data in the register.

#### 17.2.6.18 void SPI_SetDummyData ( SPI_Type * *base*, uint8_t *dummyData* )

Parameters

| | |
|---|---|
| *base* | SPI peripheral address. |
| *dummyData* | Data to be transferred when tx buffer is NULL. |

#### 17.2.6.19 void SPI_MasterTransferCreateHandle ( SPI_Type * *base*, spi_master_handle_t * *handle*, spi_master_callback_t *callback*, void * *userData* )

This function initializes the SPI master handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | SPI handle pointer. |
| *callback* | Callback function. |
| *userData* | User data. |

#### 17.2.6.20 status_t SPI_MasterTransferBlocking ( SPI_Type * *base*, spi_transfer_t * *xfer* )

Parameters

| | |
|---|---|
| *base* | SPI base pointer |
| *xfer* | pointer to spi_xfer_config_t structure |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully start a transfer. |
| *kStatus_InvalidArgument* | Input argument is invalid. |

#### 17.2.6.21 status_t SPI_MasterTransferNonBlocking ( SPI_Type ∗ *base*, spi_master_handle_t ∗ *handle*, spi_transfer_t ∗ *xfer* )

Note

The API immediately returns after transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.
If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | pointer to spi_master_handle_t structure which stores the transfer state |
| *xfer* | pointer to spi_xfer_config_t 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. |

#### 17.2.6.22 status_t SPI_MasterTransferGetCount ( SPI_Type ∗ *base*, spi_master_handle_t ∗ *handle*, size_t ∗ *count* )

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | Pointer to SPI transfer handle, this should be a static variable. |
| *count* | Transferred bytes of SPI master. |

Return values

| | |
|---|---|
| *kStatus_SPI_Success* | Succeed get the transfer count. |
| *kStatus_NoTransferInProgress* | There is not a non-blocking transaction currently in progress. |

#### 17.2.6.23 void SPI_MasterTransferAbort ( SPI_Type * *base*, spi_master_handle_t * *handle* )

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | Pointer to SPI transfer handle, this should be a static variable. |

#### 17.2.6.24 void SPI_MasterTransferHandleIRQ ( SPI_Type * *base*, spi_master_handle_t * *handle* )

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | pointer to spi_master_handle_t structure which stores the transfer state. |

#### 17.2.6.25 void SPI_SlaveTransferCreateHandle ( SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_slave_callback_t *callback*, void * *userData* )

This function initializes the SPI slave handle which can be used for other SPI slave transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | SPI handle pointer. |
| *callback* | Callback function. |
| *userData* | User data. |

#### 17.2.6.26 status_t SPI_SlaveTransferNonBlocking ( SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_transfer_t * *xfer* )

Note

The API returns immediately after the transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.
If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | pointer to spi_slave_handle_t structure which stores the transfer state |
| *xfer* | pointer to spi_xfer_config_t 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. |

#### 17.2.6.27 static status_t SPI_SlaveTransferGetCount ( SPI_Type * *base*, spi_slave_handle_t * *handle*, size_t * *count* ) `[inline]`, `[static]`

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | Pointer to SPI transfer handle, this should be a static variable. |
| *count* | Transferred bytes of SPI slave. |

Return values

| | |
|---|---|
| *kStatus_SPI_Success* | Succeed get the transfer count. |
| *kStatus_NoTransferIn-Progress* | There is not a non-blocking transaction currently in progress. |

#### 17.2.6.28 static void SPI_SlaveTransferAbort ( SPI_Type * *base*, spi_slave_handle_t * *handle* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | Pointer to SPI transfer handle, this should be a static variable. |

#### 17.2.6.29 void SPI_SlaveTransferHandleIRQ ( SPI_Type * *base*, spi_slave_handle_t * *handle* )

Parameters

| | |
|---|---|
| *base* | SPI peripheral base address. |
| *handle* | pointer to spi_slave_handle_t structure which stores the transfer state |

### 17.2.7 Variable Documentation

#### 17.2.7.1 volatile uint8_t g_spiDummyData[]

## 17.3 SPI CMSIS driver

This section describes the programming interface of the SPI Cortex Microcontroller Software Interface Standard (CMSIS) driver. And this driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage methord please refer to http://www.keil.com/pack/doc/cmsis/Driver/html/index.html.

### 17.3.1 Function groups

#### 17.3.1.1 SPI CMSIS GetVersion Operation

This function group will return the SPI CMSIS Driver version to user.

#### 17.3.1.2 SPI CMSIS GetCapabilities Operation

This function group will return the capabilities of this driver.

#### 17.3.1.3 SPI CMSIS Initialize and Uninitialize Operation

This function will initialize and uninitialize the instance in master mode or slave mode. And this API must be called before you configure an instance or after you Deinit an instance.The right steps to start an instance is that you must initialize the instance which been slected firstly,then you can power on the instance.After these all have been done,you can configure the instance by using control operation.If you want to Uninitialize the instance, you must power off the instance first.

#### 17.3.1.4 SPI CMSIS Transfer Operation

This function group controls the transfer, master send/receive data, and slave send/receive data.

#### 17.3.1.5 SPI CMSIS Status Operation

This function group gets the SPI transfer status.

#### 17.3.1.6 SPI CMSIS Control Operation

This function can configure instance as master mode or slave mode, set baudrate for master mode transfer, get current baudrate of master mode transfer,set transfer data bits and other control command.

## 17.3.2 Typical use case

### 17.3.2.1 Master Operation

```
/* Variables */
uint8_t masterRxData[TRANSFER_SIZE] = {0U};
uint8_t masterTxData[TRANSFER_SIZE] = {0U};

/*SPI master init*/
Driver_SPI0.Initialize(SPI_MasterSignalEvent_t);
Driver_SPI0.PowerControl(ARM_POWER_FULL);
Driver_SPI0.Control(ARM_SPI_MODE_MASTER, TRANSFER_BAUDRATE);

/* Start master transfer */
Driver_SPI0.Transfer(masterTxData, masterRxData, TRANSFER_SIZE);

/* Master power off */
Driver_SPI0.PowerControl(ARM_POWER_OFF);

/* Master uninitialize */
Driver_SPI0.Uninitialize();
```

### 17.3.2.2 Slave Operation

```
/* Variables */
uint8_t slaveRxData[TRANSFER_SIZE] = {0U};
uint8_t slaveTxData[TRANSFER_SIZE] = {0U};

/*SPI slave init*/
Driver_SPI1.Initialize(SPI_SlaveSignalEvent_t);
Driver_SPI1.PowerControl(ARM_POWER_FULL);
Driver_SPI1.Control(ARM_SPI_MODE_SLAVE, false);

/* Start slave transfer */
Driver_SPI1.Transfer(slaveTxData, slaveRxData, TRANSFER_SIZE);

/* slave power off */
Driver_SPI1.PowerControl(ARM_POWER_OFF);

/* slave uninitialize */
Driver_SPI1.Uninitialize();
```

# Chapter 18
# TPM: Timer PWM Module

## 18.1 Overview

The MCUXpresso SDK provides a driver for the Timer PWM Module (TPM) of MCUXpresso SDK devices.

The TPM driver supports the generation of PWM signals, input capture, and output compare modes. On some SoCs, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decoder modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoCs.

## 18.2 Introduction of TPM

### 18.2.1 Initialization and deinitialization

The function TPM_Init() initializes the TPM with a specified configurations. The function TPM_Get-DefaultConfig() gets the default configurations. On some SoCs, the initialization function issues a software reset to reset the TPM internal logic. The initialization function configures the TPM's behavior when it receives a trigger input and its operation in doze and debug modes.

The function TPM_Deinit() disables the TPM counter and turns off the module clock.

### 18.2.2 PWM Operations

The function TPM_SetupPwm() sets up TPM channels for the PWM output. The function can set up the PWM signal properties for multiple channels. Each channel has its own tpm_chnl_pwm_signal_param_t structure that is used to specify the output signals duty cycle and level-mode. However, the same PWM period and PWM mode is applied to all channels requesting a PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 where 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle). When generating a combined PWM signal, the channel number passed refers to a channel pair number, for example 0 refers to channel 0 and 1, 1 refers to channels 2 and 3.

The function TPM_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular TPM channel.

The function TPM_UpdateChnlEdgeLevelSelect() updates the level select bits of a particular TPM channel. This can be used to disable the PWM output when making changes to the PWM signal.

### 18.2.3 Input capture operations

The function TPM_SetupInputCapture() sets up a TPM channel for input capture. The user can specify the capture edge.

The function TPM_SetupDualEdgeCapture() can be used to measure the pulse width of a signal. This is available only for certain SoCs. A channel pair is used during the capture with the input signal coming through a channel that can be configured. The user can specify the capture edge for each channel and any filter value to be used when processing the input signal.

### 18.2.4 Output compare operations

The function TPM_SetupOutputCompare() sets up a TPM channel for output comparison. The user can specify the channel output on a successful comparison and a comparison value.

### 18.2.5 Quad decode

The function TPM_SetupQuadDecode() sets up TPM channels 0 and 1 for quad decode, which is available only for certain SoCs. The user can specify the quad decode mode, polarity, and filter properties for each input signal.

### 18.2.6 Fault operation

The function TPM_SetupFault() sets up the properties for each fault, which is available only for certain SoCs. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

### 18.2.7 Status

Provides functions to get and clear the TPM status.

### 18.2.8 Interrupt

Provides functions to enable/disable TPM interrupts and get current enabled interrupts.

## 18.3 Typical use case

### 18.3.1 PWM output

Output the PWM signal on 2 TPM channels with different duty cycles. Periodically update the PWM signal duty cycle. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/tpm

## Data Structures

- struct tpm_chnl_pwm_signal_param_t
  *Options to configure a TPM channel's PWM signal. More...*
- struct tpm_config_t
  *TPM config structure. More...*

## Macros

- #define TPM_MAX_COUNTER_VALUE(x) ((1U != (uint8_t)FSL_FEATURE_TPM_HAS_32BIT_COUNTERn(x)) ? 0xFFFFU : 0xFFFFFFFFU)
  *Help macro to get the max counter value.*

## Enumerations

- enum tpm_chnl_t {
  kTPM_Chnl_0 = 0U,
  kTPM_Chnl_1,
  kTPM_Chnl_2,
  kTPM_Chnl_3,
  kTPM_Chnl_4,
  kTPM_Chnl_5,
  kTPM_Chnl_6,
  kTPM_Chnl_7 }
  *List of TPM channels.*
- enum tpm_pwm_mode_t {
  kTPM_EdgeAlignedPwm = 0U,
  kTPM_CenterAlignedPwm }
  *TPM PWM operation modes.*
- enum tpm_pwm_level_select_t {
  kTPM_NoPwmSignal = 0U,
  kTPM_LowTrue,
  kTPM_HighTrue }
  *TPM PWM output pulse mode: high-true, low-true or no output.*
- enum tpm_chnl_control_bit_mask_t {
  kTPM_ChnlELSnAMask = TPM_CnSC_ELSA_MASK,
  kTPM_ChnlELSnBMask = TPM_CnSC_ELSB_MASK,
  kTPM_ChnlMSAMask = TPM_CnSC_MSA_MASK,
  kTPM_ChnlMSBMask = TPM_CnSC_MSB_MASK }
  *List of TPM channel modes and level control bit mask.*

- enum tpm_output_compare_mode_t {
kTPM_NoOutputSignal = (1U << TPM_CnSC_MSA_SHIFT),
kTPM_ToggleOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_S-HIFT)),
kTPM_ClearOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SH-IFT)),
kTPM_SetOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (3U << TPM_CnSC_ELSA_SHIF-T)),
kTPM_HighPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_-SHIFT)),
kTPM_LowPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_S-HIFT)) }
  *TPM output compare modes.*
- enum tpm_input_capture_edge_t {
kTPM_RisingEdge = (1U << TPM_CnSC_ELSA_SHIFT),
kTPM_FallingEdge = (2U << TPM_CnSC_ELSA_SHIFT),
kTPM_RiseAndFallEdge = (3U << TPM_CnSC_ELSA_SHIFT) }
  *TPM input capture edge.*
- enum tpm_clock_source_t {
kTPM_SystemClock = 1U,
kTPM_FixedClock,
kTPM_ExternalClock }
  *TPM clock source selection.*
- enum tpm_clock_prescale_t {
kTPM_Prescale_Divide_1 = 0U,
kTPM_Prescale_Divide_2,
kTPM_Prescale_Divide_4,
kTPM_Prescale_Divide_8,
kTPM_Prescale_Divide_16,
kTPM_Prescale_Divide_32,
kTPM_Prescale_Divide_64,
kTPM_Prescale_Divide_128 }
  *TPM prescale value selection for the clock source.*
- enum tpm_interrupt_enable_t {
kTPM_Chnl0InterruptEnable = (1U << 0),
kTPM_Chnl1InterruptEnable = (1U << 1),
kTPM_Chnl2InterruptEnable = (1U << 2),
kTPM_Chnl3InterruptEnable = (1U << 3),
kTPM_Chnl4InterruptEnable = (1U << 4),
kTPM_Chnl5InterruptEnable = (1U << 5),
kTPM_Chnl6InterruptEnable = (1U << 6),
kTPM_Chnl7InterruptEnable = (1U << 7),
kTPM_TimeOverflowInterruptEnable = (1U << 8) }
  *List of TPM interrupts.*
- enum tpm_status_flags_t {

kTPM_Chnl0Flag = (1U << 0),
kTPM_Chnl1Flag = (1U << 1),
kTPM_Chnl2Flag = (1U << 2),
kTPM_Chnl3Flag = (1U << 3),
kTPM_Chnl4Flag = (1U << 4),
kTPM_Chnl5Flag = (1U << 5),
kTPM_Chnl6Flag = (1U << 6),
kTPM_Chnl7Flag = (1U << 7),
kTPM_TimeOverflowFlag = (1U << 8) }
*List of TPM flags.*

## Driver version

- #define FSL_TPM_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))
  *TPM driver version 2.2.0.*

## Initialization and deinitialization

- void TPM_Init (TPM_Type ∗base, const tpm_config_t ∗config)
  *Ungates the TPM clock and configures the peripheral for basic operation.*
- void TPM_Deinit (TPM_Type ∗base)
  *Stops the counter and gates the TPM clock.*
- void TPM_GetDefaultConfig (tpm_config_t ∗config)
  *Fill in the TPM config struct with the default settings.*
- tpm_clock_prescale_t TPM_CalculateCounterClkDiv (TPM_Type ∗base, uint32_t counterPeriod_-Hz, uint32_t srcClock_Hz)
  *Calculates the counter clock prescaler.*

## Channel mode operations

- status_t TPM_SetupPwm (TPM_Type ∗base, const tpm_chnl_pwm_signal_param_t ∗chnlParams, uint8_t numOfChnls, tpm_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
  *Configures the PWM signal parameters.*
- status_t TPM_UpdatePwmDutycycle (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_pwm_-mode_t currentPwmMode, uint8_t dutyCyclePercent)
  *Update the duty cycle of an active PWM signal.*
- void TPM_UpdateChnlEdgeLevelSelect (TPM_Type ∗base, tpm_chnl_t chnlNumber, uint8_t level)
  *Update the edge level selection for a channel.*
- static uint8_t TPM_GetChannelContorlBits (TPM_Type ∗base, tpm_chnl_t chnlNumber)
  *Get the channel control bits value (mode, edge and level bit fileds).*
- static void TPM_DisableChannel (TPM_Type ∗base, tpm_chnl_t chnlNumber)
  *Dsiable the channel.*
- static void TPM_EnableChannel (TPM_Type ∗base, tpm_chnl_t chnlNumber, uint8_t control)
  *Enable the channel according to mode and level configs.*
- void TPM_SetupInputCapture (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_input_capture_-edge_t captureMode)
  *Enables capturing an input signal on the channel using the function parameters.*
- void TPM_SetupOutputCompare (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_output_-compare_mode_t compareMode, uint32_t compareValue)

*Configures the TPM to generate timed pulses.*

### Interrupt Interface

- void TPM_EnableInterrupts (TPM_Type *base, uint32_t mask)
  *Enables the selected TPM interrupts.*
- void TPM_DisableInterrupts (TPM_Type *base, uint32_t mask)
  *Disables the selected TPM interrupts.*
- uint32_t TPM_GetEnabledInterrupts (TPM_Type *base)
  *Gets the enabled TPM interrupts.*

### Status Interface

- static uint32_t TPM_GetChannelValue (TPM_Type *base, tpm_chnl_t chnlNumber)
  *Gets the TPM channel value.*
- static uint32_t TPM_GetStatusFlags (TPM_Type *base)
  *Gets the TPM status flags.*
- static void TPM_ClearStatusFlags (TPM_Type *base, uint32_t mask)
  *Clears the TPM status flags.*

### Read and write the timer period

- static void TPM_SetTimerPeriod (TPM_Type *base, uint32_t ticks)
  *Sets the timer period in units of ticks.*
- static uint32_t TPM_GetCurrentTimerCount (TPM_Type *base)
  *Reads the current timer counting value.*

### Timer Start and Stop

- static void TPM_StartTimer (TPM_Type *base, tpm_clock_source_t clockSource)
  *Starts the TPM counter.*
- static void TPM_StopTimer (TPM_Type *base)
  *Stops the TPM counter.*

## 18.4 Data Structure Documentation

### 18.4.1 struct tpm_chnl_pwm_signal_param_t

### Data Fields

- tpm_chnl_t chnlNumber
  *TPM channel to configure.*
- tpm_pwm_level_select_t level
  *PWM output active level select.*
- uint8_t dutyCyclePercent
  *PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...*

#### Field Documentation

##### (1) tpm_chnl_t tpm_chnl_pwm_signal_param_t::chnlNumber

In combined mode (available in some SoC's), this represents the channel pair number

##### (2) uint8_t tpm_chnl_pwm_signal_param_t::dutyCyclePercent

100=always active signal (100% duty cycle)

### 18.4.2 struct tpm_config_t

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

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

#### Data Fields

- tpm_clock_prescale_t prescale
  *Select TPM clock prescale value.*

## 18.5 Macro Definition Documentation

### 18.5.1 #define FSL_TPM_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))

## 18.6 Enumeration Type Documentation

### 18.6.1 enum tpm_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

***kTPM_Chnl_0*** TPM channel number 0.
***kTPM_Chnl_1*** TPM channel number 1.
***kTPM_Chnl_2*** TPM channel number 2.
***kTPM_Chnl_3*** TPM channel number 3.
***kTPM_Chnl_4*** TPM channel number 4.
***kTPM_Chnl_5*** TPM channel number 5.
***kTPM_Chnl_6*** TPM channel number 6.
***kTPM_Chnl_7*** TPM channel number 7.

### 18.6.2 enum tpm_pwm_mode_t

Enumerator

***kTPM_EdgeAlignedPwm*** Edge aligned PWM.
***kTPM_CenterAlignedPwm*** Center aligned PWM.

### 18.6.3 enum tpm_pwm_level_select_t

Note

When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Enumerator

***kTPM_NoPwmSignal*** No PWM output on pin.
***kTPM_LowTrue*** Low true pulses.
***kTPM_HighTrue*** High true pulses.

### 18.6.4 enum tpm_chnl_control_bit_mask_t

Enumerator

***kTPM_ChnlELSnAMask*** Channel ELSA bit mask.
***kTPM_ChnlELSnBMask*** Channel ELSB bit mask.
***kTPM_ChnlMSAMask*** Channel MSA bit mask.
***kTPM_ChnlMSBMask*** Channel MSB bit mask.

### 18.6.5 enum tpm_output_compare_mode_t

Enumerator

***kTPM_NoOutputSignal*** No channel output when counter reaches CnV.
***kTPM_ToggleOnMatch*** Toggle output.
***kTPM_ClearOnMatch*** Clear output.
***kTPM_SetOnMatch*** Set output.
***kTPM_HighPulseOutput*** Pulse output high.
***kTPM_LowPulseOutput*** Pulse output low.

### 18.6.6 enum tpm_input_capture_edge_t

Enumerator

***kTPM_RisingEdge*** Capture on rising edge only.
***kTPM_FallingEdge*** Capture on falling edge only.
***kTPM_RiseAndFallEdge*** Capture on rising or falling edge.

### 18.6.7 enum tpm_clock_source_t

Enumerator

***kTPM_SystemClock*** System clock.
***kTPM_FixedClock*** Fixed frequency clock.
***kTPM_ExternalClock*** External TPM_EXTCLK pin clock.

### 18.6.8 enum tpm_clock_prescale_t

Enumerator

***kTPM_Prescale_Divide_1*** Divide by 1.
***kTPM_Prescale_Divide_2*** Divide by 2.
***kTPM_Prescale_Divide_4*** Divide by 4.
***kTPM_Prescale_Divide_8*** Divide by 8.
***kTPM_Prescale_Divide_16*** Divide by 16.
***kTPM_Prescale_Divide_32*** Divide by 32.
***kTPM_Prescale_Divide_64*** Divide by 64.
***kTPM_Prescale_Divide_128*** Divide by 128.

### 18.6.9 enum tpm_interrupt_enable_t

Enumerator

***kTPM_Chnl0InterruptEnable*** Channel 0 interrupt.
***kTPM_Chnl1InterruptEnable*** Channel 1 interrupt.
***kTPM_Chnl2InterruptEnable*** Channel 2 interrupt.
***kTPM_Chnl3InterruptEnable*** Channel 3 interrupt.
***kTPM_Chnl4InterruptEnable*** Channel 4 interrupt.
***kTPM_Chnl5InterruptEnable*** Channel 5 interrupt.
***kTPM_Chnl6InterruptEnable*** Channel 6 interrupt.
***kTPM_Chnl7InterruptEnable*** Channel 7 interrupt.
***kTPM_TimeOverflowInterruptEnable*** Time overflow interrupt.

### 18.6.10 enum tpm_status_flags_t

Enumerator

***kTPM_Chnl0Flag*** Channel 0 flag.
***kTPM_Chnl1Flag*** Channel 1 flag.
***kTPM_Chnl2Flag*** Channel 2 flag.
***kTPM_Chnl3Flag*** Channel 3 flag.
***kTPM_Chnl4Flag*** Channel 4 flag.
***kTPM_Chnl5Flag*** Channel 5 flag.
***kTPM_Chnl6Flag*** Channel 6 flag.
***kTPM_Chnl7Flag*** Channel 7 flag.
***kTPM_TimeOverflowFlag*** Time overflow flag.

## 18.7 Function Documentation

### 18.7.1 void TPM_Init ( TPM_Type ∗ *base*, const tpm_config_t ∗ *config* )

Note

This API should be called at the beginning of the application using the TPM driver.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *config* | Pointer to user's TPM config structure. |

### 18.7.2 void TPM_Deinit ( TPM_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

### 18.7.3 void TPM_GetDefaultConfig ( tpm_config_t ∗ *config* )

The default values are:

```
*     config->prescale = kTPM_Prescale_Divide_1;
*     config->useGlobalTimeBase = false;
*     config->syncGlobalTimeBase = false;
*     config->dozeEnable = false;
*     config->dbgMode = false;
*     config->enableReloadOnTrigger = false;
*     config->enableStopOnOverflow = false;
```

```
*     config->enableStartOnTrigger = false;
*#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
*     config->enablePauseOnTrigger = false;
*#endif
*     config->triggerSelect = kTPM_Trigger_Select_0;
*#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
*     config->triggerSource = kTPM_TriggerSource_External;
*     config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
*#endif
*#if defined(FSL_FEATURE_TPM_HAS_POL) && FSL_FEATURE_TPM_HAS_POL
*     config->chnlPolarity = 0U;
*#endif
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to user's TPM config structure. |

### 18.7.4 tpm_clock_prescale_t TPM_CalculateCounterClkDiv ( TPM_Type ∗ *base*, uint32_t *counterPeriod_Hz*, uint32_t *srcClock_Hz* )

This function calculates the values for SC[PS].

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *counterPeriod-_Hz* | The desired frequency in Hz which corresponding to the time when the counter reaches the mod value |
| *srcClock_Hz* | TPM counter clock in Hz |

return Calculated clock prescaler value.

### 18.7.5 status_t TPM_SetupPwm ( TPM_Type ∗ *base*, const tpm_chnl_pwm_signal-_param_t ∗ *chnlParams*, uint8_t *numOfChnls*, tpm_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz* )

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

| | |
|---|---|
| *chnlParams* | Array of PWM channel parameters to configure the channel(s) |
| *numOfChnls* | Number of channels to configure, this should be the size of the array passed in |
| *mode* | PWM operation mode, options available in enumeration tpm_pwm_mode_t |
| *pwmFreq_Hz* | PWM signal frequency in Hz |
| *srcClock_Hz* | TPM counter clock in Hz |

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

### 18.7.6 status_t TPM_UpdatePwmDutycycle ( TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, tpm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent* )

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number. In combined mode, this represents the channel pair number |
| *currentPwm-Mode* | The current PWM mode set during PWM setup |
| *dutyCycle-Percent* | New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle) |

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

### 18.7.7 void TPM_UpdateChnlEdgeLevelSelect ( TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, uint8_t *level* )

Note

When the TPM has PWM pause level select feature (FSL_FEATURE_TPM_HAS_PAUSE_LEVEL_SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use TPM_DisableChannel API to close the PWM output.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |
| *level* | The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field. |

### 18.7.8 static uint8_t TPM_GetChannelContorlBits ( TPM_Type * *base,* tpm_chnl_t *chnlNumber* ) [inline], [static]

This function disable the channel by clear all mode and level control bits.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |

Returns

The contorl bits value. This is the logical OR of members of the enumeration tpm_chnl_control_bit-_mask_t.

### 18.7.9 static void TPM_DisableChannel ( TPM_Type * *base,* tpm_chnl_t *chnlNumber* ) [inline], [static]

This function disable the channel by clear all mode and level control bits.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |

### 18.7.10 static void TPM_EnableChannel ( TPM_Type * *base,* tpm_chnl_t *chnlNumber,* uint8_t *control* ) [inline], [static]

This function enable the channel output according to input mode/level config parameters.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |
| *control* | The contorl bits value. This is the logical OR of members of the enumeration tpm_chnl_control_bit_mask_t. |

### 18.7.11 void TPM_SetupInputCapture ( TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, tpm_input_capture_edge_t *captureMode* )

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |
| *captureMode* | Specifies which edge to capture |

### 18.7.12 void TPM_SetupOutputCompare ( TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, tpm_output_compare_mode_t *compareMode*, uint32_t *compareValue* )

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |
| *compareMode* | Action to take on the channel output when the compare condition is met |
| *compareValue* | Value to be programmed in the CnV register. |

### 18.7.13 void TPM_EnableInterrupts ( TPM_Type ∗ *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *mask* | The interrupts to enable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t |

### 18.7.14 void TPM_DisableInterrupts ( TPM_Type * *base*, uint32_t *mask* )

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *mask* | The interrupts to disable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t |

### 18.7.15 uint32_t TPM_GetEnabledInterrupts ( TPM_Type * *base* )

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

Returns

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

### 18.7.16 static uint32_t TPM_GetChannelValue ( TPM_Type * *base*, tpm_chnl_t *chnlNumber* ) [inline], [static]

Note

The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *chnlNumber* | The channel number |

Returns

The channle CnV regisyer value.

### 18.7.17 static uint32_t TPM_GetStatusFlags ( TPM_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

Returns

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

### 18.7.18 static void TPM_ClearStatusFlags ( TPM_Type * *base*, uint32_t *mask* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *mask* | The status flags to clear. This is a logical OR of members of the enumeration tpm_status_flags_t |

### 18.7.19 static void TPM_SetTimerPeriod ( TPM_Type * *base*, uint32_t *ticks* ) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.
2. Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *ticks* | A timer period in units of ticks, which should be equal or greater than 1. |

### 18.7.20 static uint32_t TPM_GetCurrentTimerCount ( TPM_Type * *base* ) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

Returns

The current counter value in ticks

### 18.7.21 static void TPM_StartTimer ( TPM_Type * *base*, tpm_clock_source_t *clockSource* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |
| *clockSource* | TPM clock source; once clock source is set the counter will start running |

### 18.7.22 static void TPM_StopTimer ( TPM_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | TPM peripheral base address |

# Chapter 19
# UART: Universal Asynchronous Receiver/Transmitter Driver

## 19.1 Overview

### Modules

- UART CMSIS Driver
- UART Driver

## 19.2 UART Driver

### 19.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of MCUXpresso SDK devices.

The UART driver includes functional APIs and transactional APIs.

Functional APIs are used for UART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the UART peripheral and how to organize functional APIs to meet the application requirements. All functional APIs use the peripheral base address as the first parameter. UART functional operation groups provide the functional API set.

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the uart_handle_t as the second parameter. Initialize the handle by calling the UART_TransferCreateHandle() API.

Transactional APIs support asynchronous transfer, which means that the functions UART_TransferSendNonBlocking() and UART_TransferReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_UART_TxIdle and kStatus_UART_RxIdle.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the UART_TransferCreateHandle(). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The UART_TransferReceiveNonBlocking() function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the kStatus_UART_RxIdle.

If the receive ring buffer is full, the upper layer is informed through a callback with the kStatus_UART_RxRingBufferOverrun. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code.

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart In this example, the buffer size is 32, but only 31 bytes are used for saving data.

### 19.2.2 Typical use case

#### 19.2.2.1 UART Send/receive using a polling method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart

#### 19.2.2.2 UART Send/receive using an interrupt method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart

#### 19.2.2.3 UART Receive using the ringbuffer feature

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart

#### 19.2.2.4 UART Send/Receive using the DMA method

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart

## Data Structures

- struct uart_config_t
  *UART configuration structure. More...*
- struct uart_transfer_t
  *UART transfer structure. More...*
- struct uart_handle_t
  *UART handle structure. More...*

## Macros

- #define UART_RETRY_TIMES 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */
  *Retry times for waiting flag.*

## Typedefs

- typedef void(* uart_transfer_callback_t )(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)
  *UART transfer callback function.*

## Enumerations

- enum {
kStatus_UART_TxBusy = MAKE_STATUS(kStatusGroup_UART, 0),
kStatus_UART_RxBusy = MAKE_STATUS(kStatusGroup_UART, 1),
kStatus_UART_TxIdle = MAKE_STATUS(kStatusGroup_UART, 2),
kStatus_UART_RxIdle = MAKE_STATUS(kStatusGroup_UART, 3),
kStatus_UART_TxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 4),
kStatus_UART_RxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 5),
kStatus_UART_FlagCannotClearManually,
kStatus_UART_Error = MAKE_STATUS(kStatusGroup_UART, 7),
kStatus_UART_RxRingBufferOverrun = MAKE_STATUS(kStatusGroup_UART, 8),
kStatus_UART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_UART, 9),
kStatus_UART_NoiseError = MAKE_STATUS(kStatusGroup_UART, 10),
kStatus_UART_FramingError = MAKE_STATUS(kStatusGroup_UART, 11),
kStatus_UART_ParityError = MAKE_STATUS(kStatusGroup_UART, 12),
kStatus_UART_BaudrateNotSupport,
kStatus_UART_IdleLineDetected = MAKE_STATUS(kStatusGroup_UART, 14),
kStatus_UART_Timeout = MAKE_STATUS(kStatusGroup_UART, 15) }
*Error codes for the UART driver.*
- enum uart_parity_mode_t {
kUART_ParityDisabled = 0x0U,
kUART_ParityEven = 0x2U,
kUART_ParityOdd = 0x3U }
*UART parity mode.*
- enum uart_stop_bit_count_t {
kUART_OneStopBit = 0U,
kUART_TwoStopBit = 1U }
*UART stop bit count.*
- enum uart_idle_type_select_t {
kUART_IdleTypeStartBit = 0U,
kUART_IdleTypeStopBit = 1U }
*UART idle type select.*
- enum _uart_interrupt_enable {
kUART_LinBreakInterruptEnable = (UART_BDH_LBKDIE_MASK),
kUART_RxActiveEdgeInterruptEnable = (UART_BDH_RXEDGIE_MASK),
kUART_TxDataRegEmptyInterruptEnable = (UART_C2_TIE_MASK << 8),
kUART_TransmissionCompleteInterruptEnable = (UART_C2_TCIE_MASK << 8),
kUART_RxDataRegFullInterruptEnable = (UART_C2_RIE_MASK << 8),
kUART_IdleLineInterruptEnable = (UART_C2_ILIE_MASK << 8),
kUART_RxOverrunInterruptEnable = (UART_C3_ORIE_MASK << 16),
kUART_NoiseErrorInterruptEnable = (UART_C3_NEIE_MASK << 16),
kUART_FramingErrorInterruptEnable = (UART_C3_FEIE_MASK << 16),
kUART_ParityErrorInterruptEnable = (UART_C3_PEIE_MASK << 16) }
*UART interrupt configuration structure, default settings all disabled.*
- enum {

kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),
kUART_RxOverrunFlag = (UART_S1_OR_MASK),
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),
kUART_FramingErrorFlag = (UART_S1_FE_MASK),
kUART_ParityErrorFlag = (UART_S1_PF_MASK),
kUART_LinBreakFlag,
kUART_RxActiveEdgeFlag,
kUART_RxActiveFlag }
*UART status flags.*

## Functions

- uint32_t UART_GetInstance (UART_Type ∗base)
  *Get the UART instance from peripheral base address.*

## Variables

- void ∗ s_uartHandle [ ]
  *Pointers to uart handles for each instance.*
- uart_isr_t s_uartIsr
  *Pointer to uart IRQ handler for each instance.*

## Driver version

- #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 5, 1))
  *UART driver version.*

## Initialization and deinitialization

- status_t UART_Init (UART_Type ∗base, const uart_config_t ∗config, uint32_t srcClock_Hz)
  *Initializes a UART instance with a user configuration structure and peripheral clock.*
- void UART_Deinit (UART_Type ∗base)
  *Deinitializes a UART instance.*
- void UART_GetDefaultConfig (uart_config_t ∗config)
  *Gets the default configuration structure.*
- status_t UART_SetBaudRate (UART_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  *Sets the UART instance baud rate.*
- void UART_Enable9bitMode (UART_Type ∗base, bool enable)
  *Enable 9-bit data mode for UART.*
- static void UART_Set9thTransmitBit (UART_Type ∗base)
  *Set UART 9th transmit bit.*

- static void UART_Clear9thTransmitBit (UART_Type *base)
  *Clear UART 9th transmit bit.*

### Status

- uint32_t UART_GetStatusFlags (UART_Type *base)
  *Gets UART status flags.*
- status_t UART_ClearStatusFlags (UART_Type *base, uint32_t mask)
  *Clears status flags with the provided mask.*

### Interrupts

- void UART_EnableInterrupts (UART_Type *base, uint32_t mask)
  *Enables UART interrupts according to the provided mask.*
- void UART_DisableInterrupts (UART_Type *base, uint32_t mask)
  *Disables the UART interrupts according to the provided mask.*
- uint32_t UART_GetEnabledInterrupts (UART_Type *base)
  *Gets the enabled UART interrupts.*

### Bus Operations

- static void UART_EnableTx (UART_Type *base, bool enable)
  *Enables or disables the UART transmitter.*
- static void UART_EnableRx (UART_Type *base, bool enable)
  *Enables or disables the UART receiver.*
- static void UART_WriteByte (UART_Type *base, uint8_t data)
  *Writes to the TX register.*
- static uint8_t UART_ReadByte (UART_Type *base)
  *Reads the RX register directly.*
- status_t UART_WriteBlocking (UART_Type *base, const uint8_t *data, size_t length)
  *Writes to the TX register using a blocking method.*
- status_t UART_ReadBlocking (UART_Type *base, uint8_t *data, size_t length)
  *Read RX data register using a blocking method.*

### Transactional

- void UART_TransferCreateHandle (UART_Type *base, uart_handle_t *handle, uart_transfer_callback_t callback, void *userData)
  *Initializes the UART handle.*
- void UART_TransferStartRingBuffer (UART_Type *base, uart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  *Sets up the RX ring buffer.*
- void UART_TransferStopRingBuffer (UART_Type *base, uart_handle_t *handle)
  *Aborts the background transfer and uninstalls the ring buffer.*
- size_t UART_TransferGetRxRingBufferLength (uart_handle_t *handle)

*Get the length of received data in RX ring buffer.*
- status_t UART_TransferSendNonBlocking (UART_Type ∗base, uart_handle_t ∗handle, uart_-transfer_t ∗xfer)
  *Transmits a buffer of data using the interrupt method.*
- void UART_TransferAbortSend (UART_Type ∗base, uart_handle_t ∗handle)
  *Aborts the interrupt-driven data transmit.*
- status_t UART_TransferGetSendCount (UART_Type ∗base, uart_handle_t ∗handle, uint32_t ∗count)
  *Gets the number of bytes sent out to bus.*
- status_t UART_TransferReceiveNonBlocking (UART_Type ∗base, uart_handle_t ∗handle, uart_-transfer_t ∗xfer, size_t ∗receivedBytes)
  *Receives a buffer of data using an interrupt method.*
- void UART_TransferAbortReceive (UART_Type ∗base, uart_handle_t ∗handle)
  *Aborts the interrupt-driven data receiving.*
- status_t UART_TransferGetReceiveCount (UART_Type ∗base, uart_handle_t ∗handle, uint32_-t ∗count)
  *Gets the number of bytes that have been received.*
- void UART_TransferHandleIRQ (UART_Type ∗base, void ∗irqHandle)
  *UART IRQ handle function.*
- void UART_TransferHandleErrorIRQ (UART_Type ∗base, void ∗irqHandle)
  *UART Error IRQ handle function.*

## 19.2.3 Data Structure Documentation

### 19.2.3.1 struct uart_config_t

#### Data Fields

- uint32_t baudRate_Bps
  *UART baud rate.*
- uart_parity_mode_t parityMode
  *Parity mode, disabled (default), even, odd.*
- uart_stop_bit_count_t stopBitCount
  *Number of stop bits, 1 stop bit (default) or 2 stop bits.*
- uart_idle_type_select_t idleType
  *IDLE type select.*
- bool enableTx
  *Enable TX.*
- bool enableRx
  *Enable RX.*

#### Field Documentation

##### (1) uart_idle_type_select_t uart_config_t::idleType

#### 19.2.3.2 struct uart_transfer_t

##### Data Fields

- size_t dataSize
  *The byte count to be transfer.*
- 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.*

##### Field Documentation

**(1) uint8_t∗ uart_transfer_t::data**

**(2) uint8_t∗ uart_transfer_t::rxData**

**(3) const uint8_t∗ uart_transfer_t::txData**

**(4) size_t uart_transfer_t::dataSize**

#### 19.2.3.3 struct _uart_handle

##### Data Fields

- 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.*
- uart_transfer_callback_t callback
  *Callback function.*
- void ∗ userData
  *UART callback function parameter.*

- volatile uint8_t txState
  *TX transfer state.*
- volatile uint8_t rxState
  *RX transfer state.*

**Field Documentation**

#### (1) const uint8_t* volatile uart_handle_t::txData

#### (2) volatile size_t uart_handle_t::txDataSize

#### (3) size_t uart_handle_t::txDataSizeAll

#### (4) uint8_t* volatile uart_handle_t::rxData

#### (5) volatile size_t uart_handle_t::rxDataSize

#### (6) size_t uart_handle_t::rxDataSizeAll

#### (7) uint8_t* uart_handle_t::rxRingBuffer

#### (8) size_t uart_handle_t::rxRingBufferSize

#### (9) volatile uint16_t uart_handle_t::rxRingBufferHead

#### (10) volatile uint16_t uart_handle_t::rxRingBufferTail

#### (11) uart_transfer_callback_t uart_handle_t::callback

#### (12) void* uart_handle_t::userData

#### (13) volatile uint8_t uart_handle_t::txState

## 19.2.4 Macro Definition Documentation

### 19.2.4.1 #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 5, 1))

### 19.2.4.2 #define UART_RETRY_TIMES 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */

## 19.2.5 Typedef Documentation

### 19.2.5.1 typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)

## 19.2.6 Enumeration Type Documentation

### 19.2.6.1 anonymous enum

Enumerator

***kStatus_UART_TxBusy*** Transmitter is busy.
***kStatus_UART_RxBusy*** Receiver is busy.
***kStatus_UART_TxIdle*** UART transmitter is idle.
***kStatus_UART_RxIdle*** UART receiver is idle.
***kStatus_UART_TxWatermarkTooLarge*** TX FIFO watermark too large.
***kStatus_UART_RxWatermarkTooLarge*** RX FIFO watermark too large.
***kStatus_UART_FlagCannotClearManually*** UART flag can't be manually cleared.
***kStatus_UART_Error*** Error happens on UART.
***kStatus_UART_RxRingBufferOverrun*** UART RX software ring buffer overrun.
***kStatus_UART_RxHardwareOverrun*** UART RX receiver overrun.
***kStatus_UART_NoiseError*** UART noise error.
***kStatus_UART_FramingError*** UART framing error.
***kStatus_UART_ParityError*** UART parity error.
***kStatus_UART_BaudrateNotSupport*** Baudrate is not support in current clock source.
***kStatus_UART_IdleLineDetected*** UART IDLE line detected.
***kStatus_UART_Timeout*** UART times out.

### 19.2.6.2 enum uart_parity_mode_t

Enumerator

***kUART_ParityDisabled*** Parity disabled.
***kUART_ParityEven*** Parity enabled, type even, bit setting: PE|PT = 10.
***kUART_ParityOdd*** Parity enabled, type odd, bit setting: PE|PT = 11.

### 19.2.6.3 enum uart_stop_bit_count_t

Enumerator

***kUART_OneStopBit*** One stop bit.
***kUART_TwoStopBit*** Two stop bits.

### 19.2.6.4 enum uart_idle_type_select_t

Enumerator

***kUART_IdleTypeStartBit*** Start counting after a valid start bit.
***kUART_IdleTypeStopBit*** Start counting after a stop bit.

#### 19.2.6.5 enum _uart_interrupt_enable

This structure contains the settings for all of the UART interrupt configurations.

Enumerator

***kUART_LinBreakInterruptEnable*** LIN break detect interrupt.
***kUART_RxActiveEdgeInterruptEnable*** RX active edge interrupt.
***kUART_TxDataRegEmptyInterruptEnable*** Transmit data register empty interrupt.
***kUART_TransmissionCompleteInterruptEnable*** Transmission complete interrupt.
***kUART_RxDataRegFullInterruptEnable*** Receiver data register full interrupt.
***kUART_IdleLineInterruptEnable*** Idle line interrupt.
***kUART_RxOverrunInterruptEnable*** Receiver overrun interrupt.
***kUART_NoiseErrorInterruptEnable*** Noise error flag interrupt.
***kUART_FramingErrorInterruptEnable*** Framing error flag interrupt.
***kUART_ParityErrorInterruptEnable*** Parity error flag interrupt.

#### 19.2.6.6 anonymous enum

This provides constants for the UART status flags for use in the UART functions.

Enumerator

***kUART_TxDataRegEmptyFlag*** TX data register empty flag.
***kUART_TransmissionCompleteFlag*** Transmission complete flag.
***kUART_RxDataRegFullFlag*** RX data register full flag.
***kUART_IdleLineFlag*** Idle line detect flag.
***kUART_RxOverrunFlag*** RX overrun flag.
***kUART_NoiseErrorFlag*** RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
***kUART_FramingErrorFlag*** Frame error flag, sets if logic 0 was detected where stop bit expected.
***kUART_ParityErrorFlag*** If parity enabled, sets upon parity error detection.
***kUART_LinBreakFlag*** LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
***kUART_RxActiveEdgeFlag*** RX pin active edge interrupt flag,sets when active edge detected.
***kUART_RxActiveFlag*** Receiver Active Flag (RAF), sets at beginning of valid start bit.

### 19.2.7 Function Documentation

#### 19.2.7.1 uint32_t UART_GetInstance ( UART_Type * *base* )

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

Returns

UART instance.

#### 19.2.7.2 status_t UART_Init ( UART_Type ∗ *base*, const uart_config_t ∗ *config*, uint32_t *srcClock_Hz* )

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the UART_GetDefaultConfig() function. The example below shows how to use this API to configure UART.

```
*  uart_config_t uartConfig;
*  uartConfig.baudRate_Bps = 115200U;
*  uartConfig.parityMode = kUART_ParityDisabled;
*  uartConfig.stopBitCount = kUART_OneStopBit;
*  uartConfig.txFifoWatermark = 0;
*  uartConfig.rxFifoWatermark = 1;
*  UART_Init(UART1, &uartConfig, 20000000U);
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *config* | Pointer to the user-defined configuration structure. |
| *srcClock_Hz* | UART clock source frequency in HZ. |

Return values

| | |
|---|---|
| *kStatus_UART_Baudrate-NotSupport* | Baudrate is not support in current clock source. |
| *kStatus_Success* | Status UART initialize succeed |

#### 19.2.7.3 void UART_Deinit ( UART_Type ∗ *base* )

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

#### 19.2.7.4 void UART_GetDefaultConfig ( uart_config_t ∗ *config* )

This function initializes the UART configuration structure to a default value. The default values are as follows. uartConfig->baudRate_Bps = 115200U; uartConfig->bitCountPerChar = kUART_8BitsPerChar; uartConfig->parityMode = kUART_ParityDisabled; uartConfig->stopBitCount = kUART_OneStopBit; uartConfig->txFifoWatermark = 0; uartConfig->rxFifoWatermark = 1; uartConfig->idleType = kUART_IdleTypeStartBit; uartConfig->enableTx = false; uartConfig->enableRx = false;

Parameters

| | |
|---|---|
| *config* | Pointer to configuration structure. |

#### 19.2.7.5 status_t UART_SetBaudRate ( UART_Type ∗ *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz* )

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the UART_Init.

```
*  UART_SetBaudRate(UART1, 115200U, 20000000U);
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *baudRate_Bps* | UART baudrate to be set. |
| *srcClock_Hz* | UART clock source frequency in Hz. |

Return values

| | |
|---|---|
| *kStatus_UART_Baudrate-NotSupport* | Baudrate is not support in the current clock source. |

| | |
|---|---|
| *kStatus_Success* | Set baudrate succeeded. |

#### 19.2.7.6 void UART_Enable9bitMode ( UART_Type * *base*, bool *enable* )

This function set the 9-bit mode for UART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *enable* | true to enable, flase to disable. |

#### 19.2.7.7 static void UART_Set9thTransmitBit ( UART_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

#### 19.2.7.8 static void UART_Clear9thTransmitBit ( UART_Type * *base* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

#### 19.2.7.9 uint32_t UART_GetStatusFlags ( UART_Type * *base* )

This function gets all UART status flags. The flags are returned as the logical OR value of the enumerators _uart_flags. To check a specific status, compare the return value with enumerators in _uart_flags. For example, to check whether the TX is empty, do the following.

```
*     if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
*     {
*         ...
*     }
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

Returns

UART status flags which are ORed by the enumerators in the _uart_flags.

#### 19.2.7.10 status_t UART_ClearStatusFlags ( UART_Type * *base*, uint32_t *mask* )

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. kUART_TxDataRegEmptyFlag, kUART_TransmissionCompleteFlag, kUART_RxDataRegFullFlag, kUART_RxActiveFlag, kUART_NoiseErrorInRxDataRegFlag, kUART_ParityErrorInRxDataRegFlag, kUART_TxFifoEmptyFlag,kUART_RxFifoEmptyFlag

Note

that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *mask* | The status flags to be cleared; it is logical OR value of _uart_flags. |

Return values

| | |
|---|---|
| *kStatus_UART_FlagCannotClearManually* | The flag can't be cleared by this function but it is cleared automatically by hardware. |
| *kStatus_Success* | Status in the mask is cleared. |

#### 19.2.7.11 void UART_EnableInterrupts ( UART_Type * *base*, uint32_t *mask* )

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
*     UART_EnableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *mask* | The interrupts to enable. Logical OR of _uart_interrupt_enable. |

### 19.2.7.12 void UART_DisableInterrupts ( UART_Type ∗ *base*, uint32_t *mask* )

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to disable TX empty interrupt and RX full interrupt do the following.

```
*     UART_DisableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *mask* | The interrupts to disable. Logical OR of _uart_interrupt_enable. |

### 19.2.7.13 uint32_t UART_GetEnabledInterrupts ( UART_Type ∗ *base* )

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators _uart_interrupt_enable. To check a specific interrupts enable status, compare the return value with enumerators in _uart_interrupt_enable. For example, to check whether TX empty interrupt is enabled, do the following.

```
*     uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);
*
*     if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*     {
*         ...
*     }
*
```

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

Returns

UART interrupt flags which are logical OR of the enumerators in _uart_interrupt_enable.

#### 19.2.7.14 static void UART_EnableTx ( UART_Type * *base*, bool *enable* ) [inline], [static]

This function enables or disables the UART transmitter.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *enable* | True to enable, false to disable. |

#### 19.2.7.15 static void UART_EnableRx ( UART_Type * *base*, bool *enable* ) [inline], [static]

This function enables or disables the UART receiver.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *enable* | True to enable, false to disable. |

#### 19.2.7.16 static void UART_WriteByte ( UART_Type * *base*, uint8_t *data* ) [inline], [static]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *data* | The byte to write. |

#### 19.2.7.17 static uint8_t UART_ReadByte ( UART_Type * *base* ) [inline], [static]

This function reads data from the RX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |

Returns

The byte read from UART data register.

#### 19.2.7.18 status_t UART_WriteBlocking ( UART_Type ∗ *base*, const uint8_t ∗ *data*, size_t *length* )

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* | UART peripheral base address. |
| *data* | Start address of the data to write. |
| *length* | Size of the data to write. |

Return values

| | |
|---|---|
| *kStatus_UART_Timeout* | Transmission timed out and was aborted. |
| *kStatus_Success* | Successfully wrote all data. |

#### 19.2.7.19 status_t UART_ReadBlocking ( UART_Type ∗ *base,* uint8_t ∗ *data,* size_t *length* )

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data, and reads data from the TX register.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *data* | Start address of the buffer to store the received data. |
| *length* | Size of the buffer. |

Return values

| | |
|---|---|
| *kStatus_UART_Rx-HardwareOverrun* | Receiver overrun occurred while receiving data. |
| *kStatus_UART_Noise-Error* | A noise error occurred while receiving data. |
| *kStatus_UART_Framing-Error* | A framing error occurred while receiving data. |
| *kStatus_UART_Parity-Error* | A parity error occurred while receiving data. |
| *kStatus_UART_Timeout* | Transmission timed out and was aborted. |

| | |
|---|---|
| *kStatus_Success* | Successfully received all data. |

#### 19.2.7.20 void UART_TransferCreateHandle ( UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_callback_t *callback*, void * *userData* )

This function initializes the UART handle which can be used for other UART transactional APIs. Usually, for a specified UART instance, call this API once to get the initialized handle.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |
| *callback* | The callback function. |
| *userData* | The parameter of the callback function. |

#### 19.2.7.21 void UART_TransferStartRingBuffer ( UART_Type * *base*, uart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize* )

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the UART_TransferReceiveNonBlocking() API. If data is already 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, only 31 bytes are used for saving data.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART 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. |

#### 19.2.7.22 void UART_TransferStopRingBuffer ( UART_Type ∗ *base*, uart_handle_t ∗ *handle* )

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |

#### 19.2.7.23 size_t UART_TransferGetRxRingBufferLength ( uart_handle_t * *handle* )

Parameters

| | |
|---|---|
| *handle* | UART handle pointer. |

Returns

Length of received data in RX ring buffer.

#### 19.2.7.24 status_t UART_TransferSendNonBlocking ( UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_t * *xfer* )

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 ISR, the UART driver calls the callback function and passes the kStatus_UART_TxIdle as status parameter.

Note

The kStatus_UART_TxIdle is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the kUART_-TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |
| *xfer* | UART transfer structure. See uart_transfer_t. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully start the data transmission. |
| *kStatus_UART_TxBusy* | Previous transmission still not finished; data not all written to TX register yet. |
| *kStatus_InvalidArgument* | Invalid argument. |

#### 19.2.7.25 void UART_TransferAbortSend ( UART_Type ∗ *base,* uart_handle_t ∗ *handle* )

This function aborts the interrupt-driven data sending. The user can get the remainBytes to find out how many bytes are not sent out.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |

#### 19.2.7.26 status_t UART_TransferGetSendCount ( UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uint32_t ∗ *count* )

This function gets the number of bytes sent out to bus by using the interrupt method.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |
| *count* | Send bytes count. |

Return values

| | |
|---|---|
| *kStatus_NoTransferInProgress* | No send in progress. |
| *kStatus_InvalidArgument* | The parameter is invalid. |
| *kStatus_Success* | Get successfully through the parameter `count`; |

#### 19.2.7.27 status_t UART_TransferReceiveNonBlocking ( UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*, size_t ∗ *receivedBytes* )

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 UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter kStatus_UART_RxIdle. For example, 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 UART 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* | UART peripheral base address. |
| *handle* | UART handle pointer. |
| *xfer* | UART transfer structure, see uart_transfer_t. |
| *receivedBytes* | Bytes received from the ring buffer directly. |

Return values

| | |
|---|---|
| *kStatus_Success* | Successfully queue the transfer into transmit queue. |
| *kStatus_UART_RxBusy* | Previous receive request is not finished. |
| *kStatus_InvalidArgument* | Invalid argument. |

#### 19.2.7.28 void UART_TransferAbortReceive ( UART_Type ∗ *base,* uart_handle_t ∗ *handle* )

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to know how many bytes are not received yet.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART handle pointer. |

#### 19.2.7.29 status_t UART_TransferGetReceiveCount ( UART_Type ∗ *base,* uart_handle_t ∗ *handle,* uint32_t ∗ *count* )

This function gets the number of bytes that have been received.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *handle* | UART 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`; |

#### 19.2.7.30 void UART_TransferHandleIRQ ( UART_Type * *base*, void * *irqHandle* )

This function handles the UART transmit and receive IRQ request.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *irqHandle* | UART handle pointer. |

#### 19.2.7.31 void UART_TransferHandleErrorIRQ ( UART_Type * *base*, void * *irqHandle* )

This function handles the UART error IRQ request.

Parameters

| | |
|---|---|
| *base* | UART peripheral base address. |
| *irqHandle* | UART handle pointer. |

### 19.2.8 Variable Documentation

#### 19.2.8.1 void* s_uartHandle[]

#### 19.2.8.2 uart_isr_t s_uartIsr

## 19.3 UART CMSIS Driver

This section describes the programming interface of the UART Cortex Microcontroller Software Interface Standard (CMSIS) driver. And this driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage methord see http://www.keil.com/pack/doc/cmsis/Driver/html/index.html.

The UART driver includes transactional APIs.

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements please write custom code.

### 19.3.1 UART CMSIS Driver

#### 19.3.1.1 UART Send/receive using an interrupt method

```
/* UART  callback */
void UART_Callback(uint32_t event)
{
    if (event == ARM_USART_EVENT_SEND_COMPLETE)
    {
        txBufferFull = false;
        txOnGoing = false;
    }

    if (event == ARM_USART_EVENT_RECEIVE_COMPLETE)
    {
        rxBufferEmpty = false;
        rxOnGoing = false;
    }
}
Driver_USART0.Initialize(UART_Callback);
Driver_USART0.PowerControl(ARM_POWER_FULL);
/* Send g_tipString out. */
txOnGoing = true;
Driver_USART0.Send(g_tipString, sizeof(g_tipString) - 1);

/* Wait send finished */
while (txOnGoing)
{
}
```

#### 19.3.1.2 UART Send/Receive using the DMA method

```
/* UART  callback */
void UART_Callback(uint32_t event)
{
    if (event == ARM_USART_EVENT_SEND_COMPLETE)
    {
        txBufferFull = false;
        txOnGoing = false;
    }

    if (event == ARM_USART_EVENT_RECEIVE_COMPLETE)
```

```
    {
        rxBufferEmpty = false;
        rxOnGoing = false;
    }
}

Driver_USART0.Initialize(UART_Callback);
DMAMGR_Init();
Driver_USART0.PowerControl(ARM_POWER_FULL);

/* Send g_tipString out. */
txOnGoing = true;

Driver_USART0.Send(g_tipString, sizeof(g_tipString) - 1);

/* Wait send finished */
while (txOnGoing)
{
}
```

# Chapter 20
# WDOG8: 8-bit Watchdog Timer

## 20.1 Overview

The MCUXpresso SDK provides a peripheral driver for the WDOG8 module of MCUXpresso SDK devices.

## 20.2 Typical use case

```
wdog8_config_t config;
WDOG8_GetDefaultConfig(&config);
config.timeoutValue = 0xffffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG8_Init(wdog_base,&config);
```

### WDOG8 Initialization and De-initialization

- void WDOG8_GetDefaultConfig (wdog8_config_t *config)
  *Initializes the WDOG8 configuration structure.*
- void WDOG8_Init (WDOG_Type *base, const wdog8_config_t *config)
  *Initializes the WDOG8 module.*
- void WDOG8_Deinit (WDOG_Type *base)
  *De-initializes the WDOG8 module.*

### WDOG8 functional Operation

- static void WDOG8_Enable (WDOG_Type *base)
  *Enables the WDOG8 module.*
- static void WDOG8_Disable (WDOG_Type *base)
  *Disables the WDOG8 module.*
- static void WDOG8_EnableInterrupts (WDOG_Type *base, uint8_t mask)
  *Enables the WDOG8 interrupt.*
- static void WDOG8_DisableInterrupts (WDOG_Type *base, uint8_t mask)
  *Disables the WDOG8 interrupt.*
- static uint8_t WDOG8_GetStatusFlags (WDOG_Type *base)
  *Gets the WDOG8 all status flags.*
- void WDOG8_ClearStatusFlags (WDOG_Type *base, uint8_t mask)
  *Clears the WDOG8 flag.*
- static void WDOG8_SetTimeoutValue (WDOG_Type *base, uint16_t timeoutCount)
  *Sets the WDOG8 timeout value.*
- static void WDOG8_SetWindowValue (WDOG_Type *base, uint16_t windowValue)
  *Sets the WDOG8 window value.*
- static void WDOG8_Unlock (WDOG_Type *base)
  *Unlocks the WDOG8 register written.*
- static void WDOG8_Refresh (WDOG_Type *base)
  *Refreshes the WDOG8 timer.*

- static uint16_t WDOG8_GetCounterValue (WDOG_Type *base)
  *Gets the WDOG8 counter value.*

## 20.3 Function Documentation

### 20.3.1 void WDOG8_GetDefaultConfig ( wdog8_config_t * *config* )

This function initializes the WDOG8 configuration structure to default values. The default values are:

```
*   wdog8Config->enableWdog8 = true;
*   wdog8Config->clockSource = kWDOG8_ClockSource1;
*   wdog8Config->prescaler = kWDOG8_ClockPrescalerDivide1;
*   wdog8Config->workMode.enableWait = true;
*   wdog8Config->workMode.enableStop = false;
*   wdog8Config->workMode.enableDebug = false;
*   wdog8Config->testMode = kWDOG8_TestModeDisabled;
*   wdog8Config->enableUpdate = true;
*   wdog8Config->enableInterrupt = false;
*   wdog8Config->enableWindowMode = false;
*   wdog8Config->windowValue = 0U;
*   wdog8Config->timeoutValue = 0xFFFFU;
*
```

Parameters

| | |
|---|---|
| *config* | Pointer to the WDOG8 configuration structure. |

See Also

wdog8_config_t

### 20.3.2 void WDOG8_Init ( WDOG_Type * *base*, const wdog8_config_t * *config* )

This function initializes the WDOG8. To reconfigure the WDOG8 without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
*   wdog8_config_t config;
*   WDOG8_GetDefaultConfig(&config);
*   config.timeoutValue = 0x7ffU;
*   config.enableUpdate = true;
*   WDOG8_Init(wdog_base,&config);
*
```

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |
| *config* | The configuration of the WDOG8. |

### 20.3.3 void WDOG8_Deinit ( WDOG_Type ∗ *base* )

This function shuts down the WDOG8. Ensure that the WDOG_CS1.UPDATE is 1, which means that the register update is enabled.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |

### 20.3.4 static void WDOG8_Enable ( WDOG_Type ∗ *base* ) [inline], [static]

This function writes a value into the WDOG_CS1 register to enable the WDOG8. The WDOG_CS1 register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |

### 20.3.5 static void WDOG8_Disable ( WDOG_Type ∗ *base* ) [inline], [static]

This function writes a value into the WDOG_CS1 register to disable the WDOG8. The WDOG_CS1 register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address |

### 20.3.6 static void WDOG8_EnableInterrupts ( WDOG_Type ∗ *base,* uint8_t *mask* ) [inline], [static]

This function writes a value into the WDOG_CS1 register to enable the WDOG8 interrupt. The WDOG-_CS1 register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |
| *mask* | The interrupts to enable. The parameter can be a combination of the following source if defined:<br>• kWDOG8_InterruptEnable |

### 20.3.7 static void WDOG8_DisableInterrupts ( WDOG_Type ∗ *base,* uint8_t *mask* ) [inline], [static]

This function writes a value into the WDOG_CS register to disable the WDOG8 interrupt. The WDOG-_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |
| *mask* | The interrupts to disabled. The parameter can be a combination of the following source if defined:<br>• kWDOG8_InterruptEnable |

### 20.3.8 static uint8_t WDOG8_GetStatusFlags ( WDOG_Type ∗ *base* ) [inline], [static]

This function gets all status flags.

Example to get the running flag:

```
*   uint32_t status;
*   status = WDOG8_GetStatusFlags(wdog_base) & kWDOG8_RunningFlag;
*
```

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address |

Returns

State of the status flag: asserted (true) or not-asserted (false).

See Also

_wdog8_status_flags_t
- true: related status flag has been set.
- false: related status flag is not set.

### 20.3.9 void WDOG8_ClearStatusFlags ( WDOG_Type ∗ *base,* uint8_t *mask* )

This function clears the WDOG8 status flag.

Example to clear an interrupt flag:

```
*    WDOG8_ClearStatusFlags(wdog_base,kWDOG8_InterruptFlag);
*
```

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |
| *mask* | The status flags to clear. The parameter can be any combination of the following values:<br>• kWDOG8_InterruptFlag |

### 20.3.10 static void WDOG8_SetTimeoutValue ( WDOG_Type ∗ *base,* uint16_t *timeoutCount* ) [inline], [static]

This function writes a timeout value into the WDOG_TOVALH/L register. The WDOG_TOVALH/L register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address |
| *timeoutCount* | WDOG8 timeout value, count of WDOG8 clock ticks. |

### 20.3.11 static void WDOG8_SetWindowValue ( WDOG_Type ∗ *base,* uint16_t *windowValue* ) [inline], [static]

This function writes a window value into the WDOG_WINH/L register. The WDOG_WINH/L register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |
| *windowValue* | WDOG8 window value. |

### 20.3.12 static void WDOG8_Unlock ( WDOG_Type * *base* ) [inline], [static]

This function unlocks the WDOG8 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address |

### 20.3.13 static void WDOG8_Refresh ( WDOG_Type * *base* ) [inline], [static]

This function feeds the WDOG8. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address |

### 20.3.14 static uint16_t WDOG8_GetCounterValue ( WDOG_Type * *base* ) [inline], [static]

This function gets the WDOG8 counter value.

Parameters

| | |
|---|---|
| *base* | WDOG8 peripheral base address. |

Returns

Current WDOG8 counter value.

# Chapter 21
# Debug Console

## 21.1 Overview

This chapter describes the programming interface of the debug console driver.

The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

## 21.2 Function groups

### 21.2.1 Initialization

To initialize the debug console, call the DbgConsole_Init() function with these parameters. This function automatically enables the module and the clock.

```
status_t DbgConsole_Init(uint8_t instance, uint32_t baudRate, serial_port_type_t device, uint32_t
      clkSrcFreq);
```

Selects the supported debug console hardware device type, such as

```
typedef enum _serial_port_type
{
    kSerialPort_None = 0U,
    kSerialPort_Uart = 1U,
} serial_port_type_t;
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral. The debug console state is stored in the debug_console_state_t structure, such as shown here.

```
typedef struct DebugConsoleState
{
    uint8_t uartHandleBuffer[HAL_UART_HANDLE_SIZE];
    hal_uart_status_t (*putChar)(hal_uart_handle_t handle, const uint8_t *data, size_t length);
    hal_uart_status_t (*getChar)(hal_uart_handle_t handle, uint8_t *data, size_t length);
    serial_port_type_t type;
} debug_console_state_t;
```

This example shows how to call the DbgConsole_Init() given the user configuration structure.

```
DbgConsole_Init(BOARD_DEBUG_USART_INSTANCE, BOARD_DEBUG_USART_BAUDRATE, BOARD_DEBUG_USART_TYPE,
                             BOARD_DEBUG_USART_CLK_FREQ);
```

### 21.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width][.precision][length]specifier", which is explained below

| flags | Description |
|---|---|
| - | Left-justified within the given field width. Right-justified is the default. |
| + | Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign. |
| (space) | If no sign is written, a blank space is inserted before the value. |
| # | Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed. |
| 0 | Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier). |

| Width | Description |
|---|---|
| (number) | A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger. |
| * | The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted. |

| .precision | Description |
|---|---|
| .number | For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed. |
| .* | The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted. |

| length | Description |
|---|---|
| Do not support | |

| specifier | Description |
|---|---|
| d or i | Signed decimal integer |
| f | Decimal floating point |
| F | Decimal floating point capital letters |
| x | Unsigned hexadecimal integer |
| X | Unsigned hexadecimal integer capital letters |
| o | Signed octal |
| b | Binary value |
| p | Pointer address |
| u | Unsigned decimal integer |
| c | Character |
| s | String of characters |
| n | Nothing printed |

- Support a format specifier for SCANF following this prototype " %[*][width][length]specifier", which is explained below

| * | Description |
|---|---|
| An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument. | |

| width | Description |
|---|---|
| This specifies the maximum number of characters to be read in the current reading operation. | |

| length | Description |
|---|---|
| hh | The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X). |
| h | The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X). |
| l | The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s. |
| ll | The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s. |
| L | The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G). |
| j or z or t | Not supported |

| specifier | Qualifying Input | Type of argument |
|---|---|---|
| c | Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end. | char * |
| i | Integer: : Number optionally preceded with a + or - sign | int * |
| d | Decimal integer: Number optionally preceded with a + or - sign | int * |
| a, A, e, E, f, F, g, G | Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4 | float * |
| o | Octal Integer: | int * |
| s | String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab). | char * |
| u | Unsigned decimal integer. | unsigned int * |

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE == DEBUGCONSOLE_DISABLE /* Disable debug console */
#define PRINTF
#define SCANF
#define PUTCHAR
#define GETCHAR
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_SDK /* Select printf, scanf, putchar, getchar of SDK
```

```
        version. */
#define PRINTF DbgConsole_Printf
#define SCANF DbgConsole_Scanf
#define PUTCHAR DbgConsole_Putchar
#define GETCHAR DbgConsole_Getchar
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN /* Select printf, scanf, putchar, getchar of
        toolchain. */
#define PRINTF printf
#define SCANF scanf
#define PUTCHAR putchar
#define GETCHAR getchar
#endif /* SDK_DEBUGCONSOLE */
```

### 21.2.3 SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART

There are two macros SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART added to configure PRINTF and low level output perihperal.

- The macro SDK_DEBUGCONSOLE is used for forntend. Whether debug console redirect to toolchain or SDK or disabled, it decides which is the frontend of the debug console, Tool chain or SDK. The fucntion can be set by the macro SDK_DEBUGCONSOLE.
- The macro SDK_DEBUGCONSOLE_UART is used for backend. It is use to decide whether provide low level IO implementation to toolchain printf and scanf. For example, within MCU-Xpresso, if the macro SDK_DEBUGCONSOLE_UART is defined, __sys_write and __sys_readc will be used when __REDLIB__ is defined; _write and _read will be used in other cases.The macro does not specifically refer to the perihpheral "UART". It refers to the external perihperal UART. So if the macro SDK_DEBUGCONSOLE_UART is not defined when tool-chain printf is calling, the semihosting will be used.

The following the matrix show the effects of SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_-UART on PRINTF and printf. The green mark is the default setting of the debug console.

| SDK_DEBUGCONSOLE | SDK_DEBUGCONSOLE_UART | PRINTF | printf |
|---|---|---|---|
| DEBUGCONSOLE_-REDIRECT_TO_SDK | defined | UART | UART |
| DEBUGCONSOLE_-REDIRECT_TO_SDK | undefined | UART | semihost |
| DEBUGCONSOLE_-REDIRECT_TO_TO-OLCHAIN | defined | UART | UART |
| DEBUGCONSOLE_-REDIRECT_TO_TO-OLCHAIN | undefined | semihost | semihost |
| DEBUGCONSOLE_-DISABLE | defined | No ouput | UART |
| DEBUGCONSOLE_-DISABLE | undefined | No ouput | semihost |

## 21.3 Typical use case

### Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

### Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

### Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

### Print out failure messages using MCUXpresso SDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \" %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file ,
      line, func);
    for (;;)
    {}
}
```

### Note:

To use 'printf' and 'scanf' for GNUC Base, add file **'fsl_sbrk.c'** in path: **..\{package}\devices\{subset}\utilities\fsl-_sbrk.c** to your project.

### Modules

- Semihosting

## 21.4 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism can be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

### 21.4.1 Guide Semihosting for IAR

**NOTE:** After the setting both "printf" and "scanf" are available for debugging.

#### Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

#### Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

#### Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Choose tab "General Options" -> "Library Configurations", select Semihosted, select Via semihosting. Please Make sure the SDK_DEBUGCONSOLE_UART is not defined in project settings.
4. Start the project by choosing Project>Download and Debug.
5. Choose View>Terminal I/O to display the output from the I/O operations.

### 21.4.2 Guide Semihosting for Keil µVision

**NOTE:** Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

### 21.4.3 Guide Semihosting for MCUXpresso IDE

**Step 1: Setting up the environment**

1. To set debugger options, choose Project>Properties. select the setting category.
2. Select Tool Settings, unfold MCU C Compile.
3. Select Preprocessor item.
4. Set SDK_DEBUGCONSOLE=0, if set SDK_DEBUGCONSOLE=1, the log will be redirect to the UART.

**Step 2: Building the project**

1. Compile and link the project.

**Step 3: Starting semihosting**

1. Download and debug the project.
2. When the project runs successfully, the result can be seen in the Console window.

Semihosting can also be selected through the "Quick settings" menu in the left bottom window, Quick settings->SDK Debug Console->Semihost console.

### 21.4.4 Guide Semihosting for ARMGCC

**Step 1: Setting up the environment**

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
   - "Host Name (or IP address)" : localhost
   - "Port" :2333
   - "Connection type" : Telet.
   - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

**Add to "CMakeLists.txt"**

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__heap_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__heap_size__=0x2000")

### Step 2: Building the project

1. Change "CMakeLists.txt":
   **Change** "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=nano.specs")"
   **to** "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=rdimon.specs")"
   **Replace paragraph**
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
   **To**
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --specs=rdimon.specs ")
   **Remove**
   target_link_libraries(semihosting_ARMGCC.elf debug nosys)
2. Run "build_debug.bat" to build project

### Step 3: Starting semihosting

1. Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

2. After the setting, press "enter". The PuTTY window now shows the printf() output.

# Chapter 22
# Notification Framework

## 22.1 Overview

This section describes the programming interface of the Notifier driver.

## 22.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

These are the steps for the configuration transition.

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.
   The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending a "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system switches to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This example shows how to use the Notifier in the Power Manager application.

```
#include "fsl_notifier.h"

// Definition of the Power Manager callback.
status_t callback0(notifier_notification_block_t *notify, void *data)
{

    status_t ret = kStatus_Success;

    ...
    ...
    ...

    return ret;
}
// Definition of the Power Manager user function.
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *
      userData)
```

```
{
    ...
    ...
    ...
}
...
...
...
...
...
// Main function.
int main(void)
{
    // Define a notifier handle.
    notifier_handle_t powerModeHandle;

    // Callback configuration.
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
                kNOTIFIER_CallbackBeforeAfter,
                (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    // Power mode configurations.
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    // Definition of a transition to and out the power modes.
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    // Create Notifier handle.
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
      APP_PowerModeSwitch, NULL);
    ...
    ...
    // Power mode switch.
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
      kNOTIFIER_PolicyAgreement);
}
```

## Data Structures

- struct notifier_notification_block_t
  *notification block passed to the registered callback function.* More...
- struct notifier_callback_config_t
  *Callback configuration structure.* More...
- struct notifier_handle_t
  *Notifier handle structure.* More...

## Typedefs

- typedef void notifier_user_config_t
  *Notifier user configuration type.*
- typedef status_t(∗ notifier_user_function_t )(notifier_user_config_t ∗targetConfig, void ∗userData)

*Notifier user function prototype Use this function to execute specific operations in configuration switch.*
- typedef status_t(∗ notifier_callback_t )(notifier_notification_block_t ∗notify, void ∗data)
  *Callback prototype.*

## Enumerations

- enum _notifier_status {
  kStatus_NOTIFIER_ErrorNotificationBefore,
  kStatus_NOTIFIER_ErrorNotificationAfter }
  *Notifier error codes.*
- enum notifier_policy_t {
  kNOTIFIER_PolicyAgreement,
  kNOTIFIER_PolicyForcible }
  *Notifier policies.*
- enum notifier_notification_type_t {
  kNOTIFIER_NotifyRecover = 0x00U,
  kNOTIFIER_NotifyBefore = 0x01U,
  kNOTIFIER_NotifyAfter = 0x02U }
  *Notification type.*
- enum notifier_callback_type_t {
  kNOTIFIER_CallbackBefore = 0x01U,
  kNOTIFIER_CallbackAfter = 0x02U,
  kNOTIFIER_CallbackBeforeAfter = 0x03U }
  *The callback type, which indicates kinds of notification the callback handles.*

## Functions

- status_t NOTIFIER_CreateHandle (notifier_handle_t ∗notifierHandle, notifier_user_config_t ∗∗configs, uint8_t configsNumber, notifier_callback_config_t ∗callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void ∗userData)
  *Creates a Notifier handle.*
- status_t NOTIFIER_SwitchConfig (notifier_handle_t ∗notifierHandle, uint8_t configIndex, notifier_policy_t policy)
  *Switches the configuration according to a pre-defined structure.*
- uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t ∗notifierHandle)
  *This function returns the last failed notification callback.*

# 22.3 Data Structure Documentation

## 22.3.1 struct notifier_notification_block_t

### Data Fields

- notifier_user_config_t ∗ targetConfig
  *Pointer to target configuration.*
- notifier_policy_t policy
  *Configure transition policy.*
- notifier_notification_type_t notifyType

*Configure notification type.*

#### Field Documentation

##### (1) notifier_user_config_t∗ notifier_notification_block_t::targetConfig

##### (2) notifier_policy_t notifier_notification_block_t::policy

##### (3) notifier_notification_type_t notifier_notification_block_t::notifyType

### 22.3.2 struct notifier_callback_config_t

This structure holds the configuration of callbacks. Callbacks of this type are expected to be statically allocated. This structure contains the following application-defined data. callback - pointer to the callback function callbackType - specifies when the callback is called callbackData - pointer to the data passed to the callback.

## Data Fields

- notifier_callback_t callback
  *Pointer to the callback function.*
- notifier_callback_type_t callbackType
  *Callback type.*
- void ∗ callbackData
  *Pointer to the data passed to the callback.*

#### Field Documentation

##### (1) notifier_callback_t notifier_callback_config_t::callback

##### (2) notifier_callback_type_t notifier_callback_config_t::callbackType

##### (3) void∗ notifier_callback_config_t::callbackData

### 22.3.3 struct notifier_handle_t

Notifier handle structure. Contains data necessary for the Notifier proper function. Stores references to registered configurations, callbacks, information about their numbers, user function, user data, and other internal data. NOTIFIER_CreateHandle() must be called to initialize this handle.

## Data Fields

- notifier_user_config_t ∗∗ configsTable
  *Pointer to configure table.*
- uint8_t configsNumber
  *Number of configurations.*

- notifier_callback_config_t ∗ callbacksTable
  *Pointer to callback table.*
- uint8_t callbacksNumber
  *Maximum number of callback configurations.*
- uint8_t errorCallbackIndex
  *Index of callback returns error.*
- uint8_t currentConfigIndex
  *Index of current configuration.*
- notifier_user_function_t userFunction
  *User function.*
- void ∗ userData
  *User data passed to user function.*

#### Field Documentation

**(1) notifier_user_config_t∗∗ notifier_handle_t::configsTable**

**(2) uint8_t notifier_handle_t::configsNumber**

**(3) notifier_callback_config_t∗ notifier_handle_t::callbacksTable**

**(4) uint8_t notifier_handle_t::callbacksNumber**

**(5) uint8_t notifier_handle_t::errorCallbackIndex**

**(6) uint8_t notifier_handle_t::currentConfigIndex**

**(7) notifier_user_function_t notifier_handle_t::userFunction**

**(8) void∗ notifier_handle_t::userData**

## 22.4 Typedef Documentation

### 22.4.1 typedef void notifier_user_config_t

Reference of the user defined configuration is stored in an array; the notifier switches between these configurations based on this array.

### 22.4.2 typedef status_t(∗ notifier_user_function_t)(notifier_user_config_t ∗targetConfig, void ∗userData)

Before and after this function execution, different notification is sent to registered callbacks. If this function returns any error code, NOTIFIER_SwitchConfig() exits.

Parameters

| | |
|---|---|
| *targetConfig* | target Configuration. |
| *userData* | Refers to other specific data passed to user function. |

Returns

An error code or kStatus_Success.

### 22.4.3 typedef status_t(∗ notifier_callback_t)(notifier_notification_block_t ∗notify, void ∗data)

Declaration of a callback. It is common for registered callbacks. Reference to function of this type is part of the notifier_callback_config_t callback configuration structure. Depending on callback type, function of this prototype is called (see NOTIFIER_SwitchConfig()) before configuration switch, after it or in both use cases to notify about the switch progress (see notifier_callback_type_t). When called, the type of the notification is passed as a parameter along with the reference to the target configuration structure (see notifier_notification_block_t) and any data passed during the callback registration. When notified before the configuration switch, depending on the configuration switch policy (see notifier_policy_t), the callback may deny the execution of the user function by returning an error code different than kStatus_Success (see NOTIFIER_SwitchConfig()).

Parameters

| | |
|---|---|
| *notify* | Notification block. |
| *data* | Callback data. Refers to the data passed during callback registration. Intended to pass any driver or application data such as internal state information. |

Returns

An error code or kStatus_Success.

## 22.5 Enumeration Type Documentation

### 22.5.1 enum _notifier_status

Used as return value of Notifier functions.

Enumerator

***kStatus_NOTIFIER_ErrorNotificationBefore*** An error occurs during send "BEFORE" notification.

***kStatus_NOTIFIER_ErrorNotificationAfter*** An error occurs during send "AFTER" notification.

### 22.5.2 enum notifier_policy_t

Defines whether the user function execution is forced or not. For kNOTIFIER_PolicyForcible, the user function is executed regardless of the callback results, while kNOTIFIER_PolicyAgreement policy is used to exit NOTIFIER_SwitchConfig() when any of the callbacks returns error code. See also NOTIFIER_-SwitchConfig() description.

Enumerator

***kNOTIFIER_PolicyAgreement*** NOTIFIER_SwitchConfig() method is exited when any of the callbacks returns error code.

***kNOTIFIER_PolicyForcible*** The user function is executed regardless of the results.

### 22.5.3 enum notifier_notification_type_t

Used to notify registered callbacks

Enumerator

***kNOTIFIER_NotifyRecover*** Notify IP to recover to previous work state.

***kNOTIFIER_NotifyBefore*** Notify IP that configuration setting is going to change.

***kNOTIFIER_NotifyAfter*** Notify IP that configuration setting has been changed.

### 22.5.4 enum notifier_callback_type_t

Used in the callback configuration structure (notifier_callback_config_t) to specify when the registered callback is called during configuration switch initiated by the NOTIFIER_SwitchConfig(). Callback can be invoked in following situations.

- Before the configuration switch (Callback return value can affect NOTIFIER_SwitchConfig() execution. See the NOTIFIER_SwitchConfig() and notifier_policy_t documentation).
- After an unsuccessful attempt to switch configuration
- After a successful configuration switch

Enumerator

***kNOTIFIER_CallbackBefore*** Callback handles BEFORE notification.

***kNOTIFIER_CallbackAfter*** Callback handles AFTER notification.

***kNOTIFIER_CallbackBeforeAfter*** Callback handles BEFORE and AFTER notification.

## 22.6 Function Documentation

### 22.6.1 status_t NOTIFIER_CreateHandle ( notifier_handle_t * *notifierHandle*, notifier_user_config_t ** *configs*, uint8_t *configsNumber*, notifier_callback-_config_t * *callbacks*, uint8_t *callbacksNumber*, notifier_user_function_t *userFunction*, void * *userData* )

Parameters

| | |
|---|---|
| *notifierHandle* | A pointer to the notifier handle. |
| *configs* | A pointer to an array with references to all configurations which is handled by the Notifier. |
| *configsNumber* | Number of configurations. Size of the configuration array. |
| *callbacks* | A pointer to an array of callback configurations. If there are no callbacks to register during Notifier initialization, use NULL value. |
| *callbacks-Number* | Number of registered callbacks. Size of the callbacks array. |
| *userFunction* | User function. |
| *userData* | User data passed to user function. |

Returns

An error Code or kStatus_Success.

### 22.6.2 status_t NOTIFIER_SwitchConfig ( notifier_handle_t * *notifierHandle,* uint8_t *configIndex,* notifier_policy_t *policy* )

This function sets the system to the target configuration. Before transition, the Notifier sends notifications to all callbacks registered to the callback table. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array. The same order is used for before and after switch notifications. The notifications before the configuration switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the configuration change, further execution of this function depends on the notifier policy: the configuration change is either forced (kNOTIFIER_PolicyForcible) or exited (kNOTIFIER_PolicyAgreement). When configuration change is forced, the result of the before switch notifications are ignored. If an agreement is required, if any callback returns an error code, further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked. The index of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and NOTIFIER_Get-ErrorCallback() can be used to get it. Regardless of the policies, if any callback returns an error code, an error code indicating in which phase the error occurred is returned when NOTIFIER_SwitchConfig() exits.

Parameters

| | |
|---|---|
| *notifierHandle* | pointer to notifier handle |
| *configIndex* | Index of the target configuration. |
| *policy* | Transaction policy, kNOTIFIER_PolicyAgreement or kNOTIFIER_PolicyForcible. |

Returns

An error code or kStatus_Success.

### 22.6.3 uint8_t NOTIFIER_GetErrorCallbackIndex ( notifier_handle_t * *notifierHandle* )

This function returns an index of the last callback that failed during the configuration switch while the last NOTIFIER_SwitchConfig() was called. If the last NOTIFIER_SwitchConfig() call ended successfully value equal to callbacks number is returned. The returned value represents an index in the array of static call-backs.

Parameters

| | |
|---|---|
| *notifierHandle* | Pointer to the notifier handle |

Returns

Callback Index of the last failed callback or value equal to callbacks count.

# Chapter 23
# Irq

## 23.1 Overview

### Modules

- IRQ: external interrupt (IRQ) module

### Files

- file fsl_irq.h

### Data Structures

- struct irq_config_t
  *The IRQ pin configuration structure. More...*

### Enumerations

- enum irq_edge_t {
  kIRQ_FallingEdgeorLowlevel = 0U,
  kIRQ_RisingEdgeorHighlevel = 1U }
  *Interrupt Request (IRQ) Edge Select.*
- enum irq_mode_t {
  kIRQ_DetectOnEdgesOnly = 0U,
  kIRQ_DetectOnEdgesAndEdges = 1U }
  *Interrupt Request (IRQ) Detection Mode.*

### Driver version

- #define FSL_IRQ_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
  *Version 2.0.2.*

### IRQ Configuration

- uint32_t IRQ_GetInstance (IRQ_Type ∗base)
  *Get irq instance.*
- void IRQ_Init (IRQ_Type ∗base, const irq_config_t ∗config)
  *Initializes the IRQ pin used by the board.*
- void IRQ_Deinit (IRQ_Type ∗base)
  *Deinitialize IRQ peripheral.*
- static void IRQ_Enable (IRQ_Type ∗base, bool enable)
  *Enable/disable IRQ pin.*

### IRQ interrupt Operations

- static void IRQ_EnableInterrupt (IRQ_Type ∗base, bool enable)
  *Enable/disable IRQ pin interrupt.*
- static void IRQ_ClearIRQFlag (IRQ_Type ∗base)
  *Clear IRQF flag.*
- static uint32_t IRQ_GetIRQFlag (IRQ_Type ∗base)
  *Get IRQF flag.*

## 23.2 Data Structure Documentation

### 23.2.1 struct irq_config_t

#### Data Fields

- bool enablePullDevice
  *Enable/disable the internal pullup device when the IRQ pin is enabled.*
- irq_edge_t edgeSelect
  *Select the polarity of edges or levels on the IRQ pin that cause IRQF to be set.*
- irq_mode_t detectMode
  *select either edge-only detection or edge-and-level detection*

## 23.3 Macro Definition Documentation

### 23.3.1 #define FSL_IRQ_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

## 23.4 Enumeration Type Documentation

### 23.4.1 enum irq_edge_t

Enumerator

***kIRQ_FallingEdgeorLowlevel*** IRQ is falling-edge or falling-edge/low-level sensitive.
***kIRQ_RisingEdgeorHighlevel*** IRQ is rising-edge or rising-edge/high-level sensitive.

### 23.4.2 enum irq_mode_t

Enumerator

***kIRQ_DetectOnEdgesOnly*** IRQ event is detected only on falling/rising edges.
***kIRQ_DetectOnEdgesAndEdges*** IRQ event is detected on falling/rising edges and low/high levels.

## 23.5 Function Documentation

### 23.5.1 uint32_t IRQ_GetInstance ( IRQ_Type ∗ *base* )

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer |

Return values

| | |
|---|---|
| *Irq* | instance number. |

### 23.5.2 void IRQ_Init ( IRQ_Type ∗ *base*, const irq_config_t ∗ *config* )

To initialize the IRQ pin, define a irq configuration, specify whhether enable pull-up, the edge and detect mode. Then, call the IRQ_Init() function.

This is an example to initialize irq configuration.

```
* irq_config_t config =
* {
*   true,
*   kIRQ_FallingEdgeorLowlevel,
*   kIRQ_DetectOnEdgesOnly
* }
*
```

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer |
| *config* | IRQ configuration pointer |

### 23.5.3 void IRQ_Deinit ( IRQ_Type ∗ *base* )

This function disables the IRQ clock.

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer. |

Return values

| | |
|---|---|
| *None.* | |

### 23.5.4 static void IRQ_Enable ( IRQ_Type ∗ *base*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer. |
| *enable* | true to enable IRQ pin, else disable IRQ pin. |

Return values

| | |
|---|---|
| *None.* | |

### 23.5.5 static void IRQ_EnableInterrupt ( IRQ_Type * *base*, bool *enable* ) [inline], [static]

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer. |
| *enable* | true to enable IRQF assert interrupt request, else disable. |

Return values

| | |
|---|---|
| *None.* | |

### 23.5.6 static void IRQ_ClearIRQFlag ( IRQ_Type * *base* ) [inline], [static]

This function clears the IRQF flag.

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer. |

Return values

| | |
|---|---|
| *None.* | |

### 23.5.7 static uint32_t IRQ_GetIRQFlag ( IRQ_Type * *base* ) [inline], [static]

This function returns the IRQF flag.

Parameters

| | |
|---|---|
| *base* | IRQ peripheral base pointer. |

Return values

| | |
|---|---|
| *status* | = 0 IRQF flag deasserted. = 1 IRQF flag asserted. |

# Chapter 24
# Data Structure Documentation

## 24.0.8 wdog8_config_t Struct Reference

Describes WDOG8 configuration structure.

```
#include <fsl_wdog8.h>
```

### Data Fields

- bool enableWdog8
  *Enables or disables WDOG8.*
- wdog8_clock_source_t clockSource
  *Clock source select.*
- wdog8_clock_prescaler_t prescaler
  *Clock prescaler value.*
- wdog8_work_mode_t workMode
  *Configures WDOG8 work mode in debug stop and wait mode.*
- wdog8_test_mode_t testMode
  *Configures WDOG8 test mode.*
- bool enableUpdate
  *Update write-once register enable.*
- bool enableInterrupt
  *Enables or disables WDOG8 interrupt.*
- bool enableWindowMode
  *Enables or disables WDOG8 window mode.*
- uint16_t windowValue
  *Window value.*
- uint16_t timeoutValue
  *Timeout value.*

### 24.0.8.1 Detailed Description

## 24.0.9 wdog8_work_mode_t Struct Reference

Defines WDOG8 work mode.

```
#include <fsl_wdog8.h>
```

### Data Fields

- bool enableWait

*Enables or disables WDOG8 in wait mode.*

- bool enableStop

  *Enables or disables WDOG8 in stop mode.*
- bool enableDebug

  *Enables or disables WDOG8 in debug mode.*

#### 24.0.9.1 Detailed Description

***How to Reach Us:***

**Home Page:**
nxp.com

**Web Support:**
nxp.com/support