MCUXpresso SDK API Reference Manual

NXP Semiconductors

Document Number: MCUXSDKAPIRM
Rev. 0
Jun 2019

Contents

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Contents

Section Number	Title	Page Number

Contents

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 (SDK) 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 RTOS 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 RTOS 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

Table 2: MCUXpresso SDK Folder Structure

Chapter 2
Driver errors status

- kStatus_USART_TxBusy = 5700
- kStatus_USART_RxBusy = 5701
- kStatus_USART_TxIdle = 5702
- kStatus_USART_RxIdle = 5703
- kStatus_USART_TxError = 5704
- kStatus_USART_RxError = 5705
- kStatus_USART_RxRingBufferOverrun = 5706
- kStatus_USART_NoiseError = 5707
- kStatus_USART_FramingError = 5708
- kStatus_USART_ParityError = 5709
- kStatus_USART_HardwareOverrun = 5710
- kStatus_USART_BaudrateNotSupport = 5711
- kStatus_DMA_Busy = 5000
- kStatus_I2C_Busy = 6600
- kStatus_I2C_Idle = 6601
- kStatus_I2C_Nak = 6602
- kStatus_I2C_InvalidParameter = 6603
- kStatus_I2C_BitError = 6604
- kStatus_I2C_ArbitrationLost = 6605
- kStatus_I2C_NoTransferInProgress = 6606
- kStatus_I2C_DmaRequestFail = 6607
- #kStatus_I2C_StartStopError = 6608
- #kStatus_I2C_UnexpectedState = 6609
- kStatus_I2C_Addr_Nak = 6610
- kStatus_I2C_Timeout = 6611
- kStatus_SPI_Busy = 7600
- kStatus_SPI_Idle = 7601
- kStatus_SPI_Error = 7602
- kStatus_SPI_BaudrateNotSupport = 7603
- kStatus_NOTIFIER_ErrorNotificationBefore = 9800
- kStatus_NOTIFIER_ErrorNotificationAfter = 9801

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

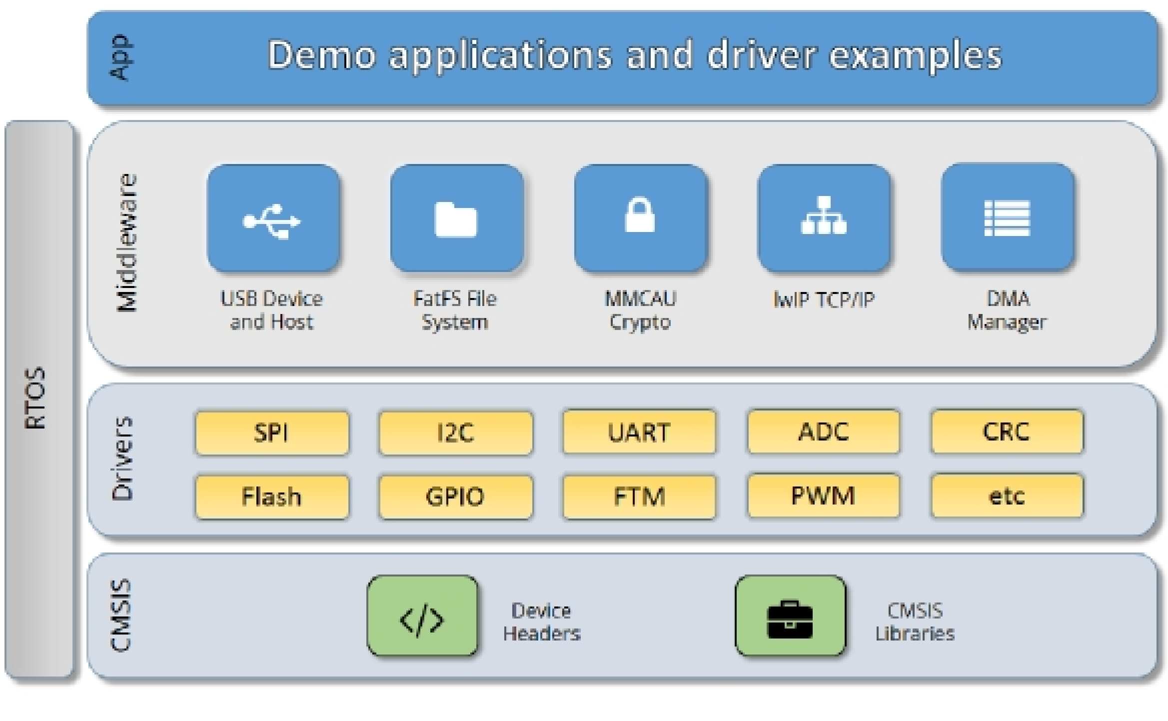

Figure 1: MCUXpresso SDK Block Diagram

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/<DEVICE_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 MCUXpresso 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
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: http://www.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, Dynam-IQ, 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 5
Common Driver

5.1 Overview

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

Macros

- #define MAKE_STATUS(group, code) ((((group)*100) + (code)))
  Construct a status code value from a group and code number.
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (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 ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
  Computes the number of elements in an array.
- #define FLASH_RSTS_N

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_CODEC = 148 }
Status group numbers.
- enum _generic_status
Generic status return codes.
- enum SYSCON_RSTn_t {
kFLASH_RST_N_SHIFT_RSTn = 0 | 4U,
kI2C0_RST_N_SHIFT_RSTn = 0 | 5U,
kGPIO0_RST_N_SHIFT_RSTn = 0 | 6U,
kSWM_RST_N_SHIFT_RSTn = 0 | 7U,
kSCT_RST_N_SHIFT_RSTn = 0 | 8U,
kWKT_RST_N_SHIFT_RSTn = 0 | 9U,
kMRT_RST_N_SHIFT_RSTn = 0 | 10U,
kSPI0_RST_N_SHIFT_RSTn = 0 | 11U,
kSPI1_RST_N_SHIFT_RSTn = 0 | 12U,
kCRC_RST_SHIFT_RSTn = 0 | 13U,
kUART0_RST_N_SHIFT_RSTn = 0 | 14U,
kUART1_RST_N_SHIFT_RSTn = 0 | 15U,
kUART2_RST_N_SHIFT_RSTn = 0 | 16U,
kIOCON_RST_N_SHIFT_RSTn = 0 | 18U,
kACMP_RST_N_SHIFT_RSTn = 0 | 19U,
kGPIO1_RST_N_SHIFT_RSTn = 0 | 20U,
kI2C1_RST_N_SHIFT_RSTn = 0 | 21U,
kI2C2_RST_N_SHIFT_RSTn = 0 | 22U,
kI2C3_RST_N_SHIFT_RSTn = 0 | 23U,
kADC_RST_N_SHIFT_RSTn = 0 | 24U,
kCTIMER0_RST_N_SHIFT_RSTn = 0 | 25U,
kDAC0_RST_N_SHIFT_RSTn = 0 | 27U,
kGPIOINT_RST_N_SHIFT_RSTn = 0 | 28U,
kDMA_RST_N_SHIFT_RSTn = 0 | 29U,
kUART3_RST_N_SHIFT_RSTn = 0 | 30U,
kUART4_RST_N_SHIFT_RSTn = 0 | 31U,
kCAPT_RST_N_SHIFT_RSTn = 65536 | 0U,
kDAC1_RST_N_SHIFT_RSTn = 65536 | 1U,
kFRG0_RST_N_SHIFT_RSTn = 65536 | 3U,
kFRG1_RST_N_SHIFT_RSTn = 65536 | 4U }
Enumeration for peripheral reset control bits.

Functions

- static status_t EnableIRQ (IRQn_Type interrupt)
Enable specific interrupt.
- static status_t DisableIRQ (IRQn_Type interrupt)
Disable specific interrupt.
- static uint32_t DisableGlobalIRQ (void)
Disable the global IRQ.
- static void EnableGlobalIRQ (uint32_t primask)
Enable the global IRQ.

- 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 RESET_PeripheralReset (reset_ip_name_t peripheral)
  Reset peripheral module.

Driver version

- #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  common driver version 2.0.1.

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)

Timer utilities

- #define USEC_TO_COUNT(us, clockFreqInHz) (uint64_t)((uint64_t)us ∗ clockFreqInHz / 1000000U)
  Macro to convert a microsecond period to raw count value.
- #define COUNT_TO_USEC(count, clockFreqInHz) (uint64_t)((uint64_t)count ∗ 1000000U / clockFreqInHz)
  Macro to convert a raw count value to microsecond.
- #define MSEC_TO_COUNT(ms, clockFreqInHz) (uint64_t)((uint64_t)ms ∗ clockFreqInHz / 1000U)
  Macro to convert a millisecond period to raw count value.
- #define COUNT_TO_MSEC(count, clockFreqInHz) (uint64_t)((uint64_t)count ∗ 1000U / clockFreqInHz)
  Macro to convert a raw count value to millisecond.

Alignment variable definition macros

- #define **SDK_ALIGN**(var, alignbytes) var
- #define SDK_SIZEALIGN(var, alignbytes) ((unsigned int)((var) + ((alignbytes)-1)) & (unsigned int)(∼(unsigned int)((alignbytes)-1)))
  Macro to change a value to a given size aligned value.

Non-cacheable region definition macros

- #define **AT_NONCACHEABLE_SECTION**(var) var
- #define **AT_NONCACHEABLE_SECTION_ALIGN**(var, alignbytes) var
- #define **AT_NONCACHEABLE_SECTION_INIT**(var) var
- #define **AT_NONCACHEABLE_SECTION_ALIGN_INIT**(var, alignbytes) var

Driver version

- #define FSL_RESET_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  reset driver version 2.0.1.

5.2 Macro Definition Documentation

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

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

5.2.3 #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

5.2.4 #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U

5.2.5 #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U

5.2.6 #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U

5.2.7 #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U

5.2.8 #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U

5.2.9 #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U

5.2.10 #define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U

5.2.11 #define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U

5.2.12 #define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U

5.2.13 #define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U

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

5.2.15 #define FSL_RESET_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

5.2.16 #define FLASH_RSTS_N

Value:

```
{                            \
        kFLASH_RST_N_SHIFT_RSTn \
    } /* Reset bits for Flash peripheral */
```

Array initializers with peripheral reset bits

5.3 Typedef Documentation

5.3.1 typedef int32_t status_t

5.4 Enumeration Type Documentation

5.4.1 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_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_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.

5.4.2 enum _generic_status

5.4.3 enum SYSCON_RSTn_t

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

Enumerator

kFLASH_RST_N_SHIFT_RSTn Flash controller reset control
kI2C0_RST_N_SHIFT_RSTn I2C0 reset control
kGPIO0_RST_N_SHIFT_RSTn GPIO0 reset control
kSWM_RST_N_SHIFT_RSTn SWM reset control
kSCT_RST_N_SHIFT_RSTn SCT reset control
kWKT_RST_N_SHIFT_RSTn Self-wake-up timer(WKT) reset control
kMRT_RST_N_SHIFT_RSTn Multi-rate timer(MRT) reset control
kSPI0_RST_N_SHIFT_RSTn SPI0 reset control.
kSPI1_RST_N_SHIFT_RSTn SPI1 reset control
kCRC_RST_SHIFT_RSTn CRC reset control
kUART0_RST_N_SHIFT_RSTn UART0 reset control
kUART1_RST_N_SHIFT_RSTn UART1 reset control
kUART2_RST_N_SHIFT_RSTn UART2 reset control
kIOCON_RST_N_SHIFT_RSTn IOCON reset control
kACMP_RST_N_SHIFT_RSTn Analog comparator reset control
kGPIO1_RST_N_SHIFT_RSTn GPIO1 reset control
kI2C1_RST_N_SHIFT_RSTn I2C1 reset control

kI2C2_RST_N_SHIFT_RSTn I2C2 reset control
kI2C3_RST_N_SHIFT_RSTn I2C3 reset control
kADC_RST_N_SHIFT_RSTn ADC reset control
kCTIMER0_RST_N_SHIFT_RSTn CTIMER0 reset control
kDAC0_RST_N_SHIFT_RSTn DAC0 reset control
kGPIOINT_RST_N_SHIFT_RSTn GPIOINT reset control
kDMA_RST_N_SHIFT_RSTn DMA reset control
kUART3_RST_N_SHIFT_RSTn UART3 reset control
kUART4_RST_N_SHIFT_RSTn UART4 reset control
kCAPT_RST_N_SHIFT_RSTn Capacitive Touch reset control
kDAC1_RST_N_SHIFT_RSTn DAC1 reset control
kFRG0_RST_N_SHIFT_RSTn Fractional baud rate generator 0 reset control
kFRG1_RST_N_SHIFT_RSTn Fractional baud rate generator 1 reset control

5.5 Function Documentation

5.5.1 static status_t EnableIRQ (IRQn_Type *interrupt*) [inline], [static]

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

interrupt	The IRQ number.

Return values

kStatus_Success	Interrupt enabled successfully
kStatus_Fail	Failed to enable the interrupt

5.5.2 static status_t DisableIRQ (IRQn_Type *interrupt*) [inline], [static]

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

interrupt	The IRQ number.

Return values

kStatus_Success	Interrupt disabled successfully
kStatus_Fail	Failed to disable the interrupt

5.5.3 static uint32_t DisableGlobalIRQ (void) [inline], [static]

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

5.5.4 static void EnableGlobalIRQ (uint32_t *primask*) [inline], [static]

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

primask	value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

5.5.5 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.

5.5.6 void SDK_Free (void * *ptr*)

Parameters

ptr	The memory to be release.

5.5.7 void RESET_PeripheralReset (reset_ip_name_t *peripheral*)

Reset peripheral module.

Parameters

peripheral	Peripheral to reset. The enum argument contains encoding of reset register and reset bit position in the reset register.

Chapter 6
USART: Universal Asynchronous Receiver/Transmitter Driver

6.1 Overview

The MCUXpresso SDK provides a peripheral USART driver for the Universal Synchronous Receiver/-Transmitter (USART) module of MCUXpresso SDK devices. The driver does not support synchronous mode.

The USART driver includes two parts: functional APIs and transactional APIs.

Functional APIs are used for USART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the USART peripheral and know how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. USART functional operation groups provide the functional APIs 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 usart_handle_t as the second parameter. Initialize the handle by calling the USART_Transfer-CreateHandle() API.

Transactional APIs support asynchronous transfer, which means that the functions USART_TransferSend-NonBlocking() and USART_TransferReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_USART_-TxIdle and kStatus_USART_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 USART_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 USART_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_USART_RxIdle.

If the receive ring buffer is full, the upper layer is informed through a callback with the kStatus_USAR-T_RxRingBufferOverrun. In the callback function, the upper layer reads data out from the ring buffer. If not, the oldest 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/usart In this example, the buffer size is 32, but only 31 bytes are used for saving data.

6.2 Typical use case

6.2.1 USART Send/receive using a polling method

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

6.2.2 USART Send/receive using an interrupt method

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

6.2.3 USART Receive using the ringbuffer feature

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

6.2.4 USART Send/Receive using the DMA method

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

Modules

- USART Driver

6.3 USART Driver

6.3.1 Overview

Data Structures

- struct usart_config_t
  USART configuration structure. More...
- struct usart_transfer_t
  USART transfer structure. More...
- struct usart_handle_t
  USART handle structure. More...

Macros

- #define FSL_SDK_ENABLE_USART_DRIVER_TRANSACTIONAL_APIS 1
  Macro gate for enable transaction API.
- #define FSL_SDK_USART_DRIVER_ENABLE_BAUDRATE_AUTO_GENERATE 1
  USART baud rate auto generate switch gate.

Typedefs

- typedef void(∗ usart_transfer_callback_t)(USART_Type ∗base, usart_handle_t ∗handle, status_t status, void ∗userData)
  USART transfer callback function.

Enumerations

- enum _usart_status {
  kStatus_USART_TxBusy = MAKE_STATUS(kStatusGroup_LPC_USART, 0),
  kStatus_USART_RxBusy = MAKE_STATUS(kStatusGroup_LPC_USART, 1),
  kStatus_USART_TxIdle = MAKE_STATUS(kStatusGroup_LPC_USART, 2),
  kStatus_USART_RxIdle = MAKE_STATUS(kStatusGroup_LPC_USART, 3),
  kStatus_USART_TxError = MAKE_STATUS(kStatusGroup_LPC_USART, 4),
  kStatus_USART_RxError = MAKE_STATUS(kStatusGroup_LPC_USART, 5),
  kStatus_USART_RxRingBufferOverrun = MAKE_STATUS(kStatusGroup_LPC_USART, 6),
  kStatus_USART_NoiseError = MAKE_STATUS(kStatusGroup_LPC_USART, 7),
  kStatus_USART_FramingError = MAKE_STATUS(kStatusGroup_LPC_USART, 8),
  kStatus_USART_ParityError = MAKE_STATUS(kStatusGroup_LPC_USART, 9),
  kStatus_USART_HardwareOverrun = MAKE_STATUS(kStatusGroup_LPC_USART, 10),
  kStatus_USART_BaudrateNotSupport }
  Macro for generating baud rate manually.

- enum usart_parity_mode_t {
  kUSART_ParityDisabled = 0x0U,
  kUSART_ParityEven = 0x2U,
  kUSART_ParityOdd = 0x3U }
  USART parity mode.
- enum usart_sync_mode_t {
  kUSART_SyncModeDisabled = 0x0U,
  kUSART_SyncModeSlave = 0x2U,
  kUSART_SyncModeMaster = 0x3U }
  USART synchronous mode.
- enum usart_stop_bit_count_t {
  kUSART_OneStopBit = 0U,
  kUSART_TwoStopBit = 1U }
  USART stop bit count.
- enum usart_data_len_t {
  kUSART_7BitsPerChar = 0U,
  kUSART_8BitsPerChar = 1U }
  USART data size.
- enum usart_clock_polarity_t {
  kUSART_RxSampleOnFallingEdge = 0x0U,
  kUSART_RxSampleOnRisingEdge = 0x1U }
  USART clock polarity configuration, used in sync mode.
- enum _usart_interrupt_enable {
  kUSART_RxReadyInterruptEnable = (USART_INTENSET_RXRDYEN_MASK),
  kUSART_TxReadyInterruptEnable = (USART_INTENSET_TXRDYEN_MASK),
  kUSART_TxIdleInterruptEnable = (USART_INTENSET_TXIDLEEN_MASK),
  kUSART_DeltaCtsInterruptEnable = (USART_INTENSET_DELTACTSEN_MASK),
  kUSART_TxDisableInterruptEnable = (USART_INTENSET_TXDISEN_MASK),
  kUSART_HardwareOverRunInterruptEnable = (USART_INTENSET_OVERRUNEN_MASK),
  kUSART_RxBreakInterruptEnable = (USART_INTENSET_DELTARXBRKEN_MASK),
  kUSART_RxStartInterruptEnable = (USART_INTENSET_STARTEN_MASK),
  kUSART_FramErrorInterruptEnable = (USART_INTENSET_FRAMERREN_MASK),
  kUSART_ParityErrorInterruptEnable = (USART_INTENSET_PARITYERREN_MASK),
  kUSART_RxNoiseInterruptEnable = (USART_INTENSET_RXNOISEEN_MASK),
  kUSART_AutoBaudErrorInterruptEnable = (USART_INTENSET_ABERREN_MASK),
  kUSART_AllInterruptEnable }
  USART interrupt configuration structure, default settings all disabled.
- enum _usart_flags {

kUSART_RxReady = (USART_STAT_RXRDY_MASK),
kUSART_RxIdleFlag = (USART_STAT_RXIDLE_MASK),
kUSART_TxReady = (USART_STAT_TXRDY_MASK),
kUSART_TxIdleFlag = (USART_STAT_TXIDLE_MASK),
kUSART_CtsState = (USART_STAT_CTS_MASK),
kUSART_DeltaCtsFlag = (USART_STAT_DELTACTS_MASK),
kUSART_TxDisableFlag = (USART_STAT_TXDISSTAT_MASK),
kUSART_HardwareOverrunFlag = (USART_STAT_OVERRUNINT_MASK),
kUSART_RxBreakFlag = (USART_STAT_DELTARXBRK_MASK),
kUSART_RxStartFlag = (USART_STAT_START_MASK),
kUSART_FramErrorFlag = (USART_STAT_FRAMERRINT_MASK),
kUSART_ParityErrorFlag = (USART_STAT_PARITYERRINT_MASK),
kUSART_RxNoiseFlag = (USART_STAT_RXNOISEINT_MASK),
kUSART_AutoBaudErrorFlag = (USART_STAT_ABERR_MASK) }
    USART status flags.

Driver version

- #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
    USART driver version 2.1.0.

Get the instance of USART

- uint32_t USART_GetInstance (USART_Type ∗base)
    Returns instance number for USART peripheral base address.

Initialization and deinitialization

- status_t USART_Init (USART_Type ∗base, const usart_config_t ∗config, uint32_t srcClock_Hz)
    Initializes a USART instance with user configuration structure and peripheral clock.
- void USART_Deinit (USART_Type ∗base)
    Deinitializes a USART instance.
- void USART_GetDefaultConfig (usart_config_t ∗config)
    Gets the default configuration structure.
- status_t USART_SetBaudRate (USART_Type ∗base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
    Sets the USART instance baud rate.

Status

- static uint32_t USART_GetStatusFlags (USART_Type ∗base)
    Get USART status flags.
- static void USART_ClearStatusFlags (USART_Type ∗base, uint32_t mask)

Clear USART status flags.

Interrupts

- static void USART_EnableInterrupts (USART_Type *base, uint32_t mask)
  Enables USART interrupts according to the provided mask.
- static void USART_DisableInterrupts (USART_Type *base, uint32_t mask)
  Disables USART interrupts according to a provided mask.
- static uint32_t USART_GetEnabledInterrupts (USART_Type *base)
  Returns enabled USART interrupts.

Bus Operations

- static void USART_EnableContinuousSCLK (USART_Type *base, bool enable)
  Continuous Clock generation.
- static void USART_EnableAutoClearSCLK (USART_Type *base, bool enable)
  Enable Continuous Clock generation bit auto clear.
- static void USART_EnableCTS (USART_Type *base, bool enable)
  Enable CTS.
- static void USART_EnableTx (USART_Type *base, bool enable)
  Enable the USART transmit.
- static void USART_EnableRx (USART_Type *base, bool enable)
  Enable the USART receive.
- static void USART_WriteByte (USART_Type *base, uint8_t data)
  Writes to the TXDAT register.
- static uint8_t USART_ReadByte (USART_Type *base)
  Reads the RXDAT directly.
- void USART_WriteBlocking (USART_Type *base, const uint8_t *data, size_t length)
  Writes to the TX register using a blocking method.
- status_t USART_ReadBlocking (USART_Type *base, uint8_t *data, size_t length)
  Read RX data register using a blocking method.

Transactional

- status_t USART_TransferCreateHandle (USART_Type *base, usart_handle_t *handle, usart_transfer_callback_t callback, void *userData)
  Initializes the USART handle.
- status_t USART_TransferSendNonBlocking (USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void USART_TransferStartRingBuffer (USART_Type *base, usart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void USART_TransferStopRingBuffer (USART_Type *base, usart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- size_t USART_TransferGetRxRingBufferLength (usart_handle_t *handle)
  Get the length of received data in RX ring buffer.

- void USART_TransferAbortSend (USART_Type ∗base, usart_handle_t ∗handle)
  Aborts the interrupt-driven data transmit.
- status_t USART_TransferGetSendCount (USART_Type ∗base, usart_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been written to USART TX register.
- status_t USART_TransferReceiveNonBlocking (USART_Type ∗base, usart_handle_t ∗handle, usart_transfer_t ∗xfer, size_t ∗receivedBytes)
  Receives a buffer of data using an interrupt method.
- void USART_TransferAbortReceive (USART_Type ∗base, usart_handle_t ∗handle)
  Aborts the interrupt-driven data receiving.
- status_t USART_TransferGetReceiveCount (USART_Type ∗base, usart_handle_t ∗handle, uint32-_t ∗count)
  Get the number of bytes that have been received.
- void USART_TransferHandleIRQ (USART_Type ∗base, usart_handle_t ∗handle)
  USART IRQ handle function.

6.3.2 Data Structure Documentation

6.3.2.1 struct usart_config_t

Data Fields

- uint32_t baudRate_Bps
  USART baud rate.
- bool enableRx
  USART receive enable.
- bool enableTx
  USART transmit enable.
- bool loopback
  Enable peripheral loopback.
- bool enableContinuousSCLK
  USART continuous Clock generation enable in synchronous master mode.
- usart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- usart_stop_bit_count_t stopBitCount
  Number of stop bits, 1 stop bit (default) or 2 stop bits.
- usart_data_len_t bitCountPerChar
  Data length - 7 bit, 8 bit.
- usart_sync_mode_t syncMode
  Transfer mode - asynchronous, synchronous master, synchronous slave.
- usart_clock_polarity_t clockPolarity
  Selects the clock polarity and sampling edge in sync mode.

6.3.2.1.0.1 Field Documentation

6.3.2.1.0.1.1 bool usart_config_t::enableRx

6.3.2.1.0.1.2 bool usart_config_t::enableTx

6.3.2.1.0.1.3 bool usart_config_t::enableContinuousSCLK

6.3.2.1.0.1.4 usart_sync_mode_t usart_config_t::syncMode

6.3.2.1.0.1.5 usart_clock_polarity_t usart_config_t::clockPolarity

6.3.2.2 struct usart_transfer_t

Data Fields

- uint8_t * data
  The buffer of data to be transfer.
- size_t dataSize
  The byte count to be transfer.

6.3.2.2.0.2 Field Documentation

6.3.2.2.0.2.1 uint8_t* usart_transfer_t::data

6.3.2.2.0.2.2 size_t usart_transfer_t::dataSize

6.3.2.3 struct _usart_handle

Data Fields

- uint8_t *volatile txData
  Address of remaining data to send.
- volatile size_t txDataSize
  Size of the remaining data to send.
- size_t txDataSizeAll
  Size of the data to send out.
- uint8_t *volatile rxData
  Address of remaining data to receive.
- volatile size_t rxDataSize
  Size of the remaining data to receive.
- size_t rxDataSizeAll
  Size of the data to receive.
- uint8_t * rxRingBuffer
  Start address of the receiver ring buffer.
- size_t rxRingBufferSize
  Size of the ring buffer.
- volatile uint16_t rxRingBufferHead
  Index for the driver to store received data into ring buffer.
- volatile uint16_t rxRingBufferTail
  Index for the user to get data from the ring buffer.

- usart_transfer_callback_t callback
  Callback function.
- void ∗ userData
  USART callback function parameter.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

6.3.2.3.0.3 Field Documentation

6.3.2.3.0.3.1 uint8_t∗ volatile usart_handle_t::txData

6.3.2.3.0.3.2 volatile size_t usart_handle_t::txDataSize

6.3.2.3.0.3.3 size_t usart_handle_t::txDataSizeAll

6.3.2.3.0.3.4 uint8_t∗ volatile usart_handle_t::rxData

6.3.2.3.0.3.5 volatile size_t usart_handle_t::rxDataSize

6.3.2.3.0.3.6 size_t usart_handle_t::rxDataSizeAll

6.3.2.3.0.3.7 uint8_t∗ usart_handle_t::rxRingBuffer

6.3.2.3.0.3.8 size_t usart_handle_t::rxRingBufferSize

6.3.2.3.0.3.9 volatile uint16_t usart_handle_t::rxRingBufferHead

6.3.2.3.0.3.10 volatile uint16_t usart_handle_t::rxRingBufferTail

6.3.2.3.0.3.11 usart_transfer_callback_t usart_handle_t::callback

6.3.2.3.0.3.12 void∗ usart_handle_t::userData

6.3.2.3.0.3.13 volatile uint8_t usart_handle_t::txState

6.3.3 Macro Definition Documentation

6.3.3.1 #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

6.3.3.2 #define FSL_SDK_ENABLE_USART_DRIVER_TRANSACTIONAL_APIS 1

1 for enable, 0 for disable.

6.3.3.3 #define FSL_SDK_USART_DRIVER_ENABLE_BAUDRATE_AUTO_GENERATE 1

1 for enable, 0 for disable

6.3.4 Typedef Documentation

6.3.4.1 typedef void(∗ usart_transfer_callback_t)(USART_Type ∗base, usart_handle_t ∗handle, status_t status, void ∗userData)

6.3.5 Enumeration Type Documentation

6.3.5.1 enum _usart_status

```
 Table of common register values for generating baud rate in specific USART clock freq
    Target baud rate(Hz)  |  USART clock frequency(Hz)   |      OSR value      |    BR
        9600              |         12,000,000           |       10            |
        9600              |         24,000,000           |       10            |
        9600              |         30,000,000           |       16            |
        9600              |         12,000,000           | NO OSR register(16) |
        115200            |         12,000,000           |       13            |
        115200            |         24,000,000           |       16            |
        115200            |         30,000,000           |       13            |
```

```
Note
```

```
: The formula for generating a baud rate is: baduRate = usartClock_Hz / (OSR * (BRG +1
       For some devices, there is no OSR register for setting, so the default OSR valu
       If the USART clock source can not generate a precise baud rate, please setting
       in SYSCON module to get a precise USART clock frequency.
```

```
Error codes for the USART driver.
```

Enumerator

kStatus_USART_TxBusy Transmitter is busy.
kStatus_USART_RxBusy Receiver is busy.
kStatus_USART_TxIdle USART transmitter is idle.
kStatus_USART_RxIdle USART receiver is idle.
kStatus_USART_TxError Error happens on tx.
kStatus_USART_RxError Error happens on rx.
kStatus_USART_RxRingBufferOverrun Error happens on rx ring buffer.
kStatus_USART_NoiseError USART noise error.
kStatus_USART_FramingError USART framing error.
kStatus_USART_ParityError USART parity error.
kStatus_USART_HardwareOverrun USART hardware over flow.
kStatus_USART_BaudrateNotSupport Baudrate is not support in current clock source.

6.3.5.2 enum usart_parity_mode_t

Enumerator

kUSART_ParityDisabled Parity disabled.
kUSART_ParityEven Parity enabled, type even, bit setting: PARITYSEL = 10.
kUSART_ParityOdd Parity enabled, type odd, bit setting: PARITYSEL = 11.

6.3.5.3 enum usart_sync_mode_t

Enumerator

kUSART_SyncModeDisabled Asynchronous mode.
kUSART_SyncModeSlave Synchronous slave mode.
kUSART_SyncModeMaster Synchronous master mode.

6.3.5.4 enum usart_stop_bit_count_t

Enumerator

kUSART_OneStopBit One stop bit.
kUSART_TwoStopBit Two stop bits.

6.3.5.5 enum usart_data_len_t

Enumerator

kUSART_7BitsPerChar Seven bit mode.
kUSART_8BitsPerChar Eight bit mode.

6.3.5.6 enum usart_clock_polarity_t

Enumerator

kUSART_RxSampleOnFallingEdge Un_RXD is sampled on the falling edge of SCLK.
kUSART_RxSampleOnRisingEdge Un_RXD is sampled on the rising edge of SCLK.

6.3.5.7 enum _usart_interrupt_enable

Enumerator

kUSART_RxReadyInterruptEnable Receive ready interrupt.

kUSART_TxReadyInterruptEnable Transmit ready interrupt.
kUSART_TxIdleInterruptEnable Transmit idle interrupt.
kUSART_DeltaCtsInterruptEnable Cts pin change interrupt.
kUSART_TxDisableInterruptEnable Transmit disable interrupt.
kUSART_HardwareOverRunInterruptEnable hardware ove run interrupt.
kUSART_RxBreakInterruptEnable Receive break interrupt.
kUSART_RxStartInterruptEnable Receive ready interrupt.
kUSART_FramErrorInterruptEnable Receive start interrupt.
kUSART_ParityErrorInterruptEnable Receive frame error interrupt.
kUSART_RxNoiseInterruptEnable Receive noise error interrupt.
kUSART_AutoBaudErrorInterruptEnable Receive auto baud error interrupt.
kUSART_AllInterruptEnable All interrupt.

6.3.5.8 enum _usart_flags

This provides constants for the USART status flags for use in the USART functions.

Enumerator

kUSART_RxReady Receive ready flag.
kUSART_RxIdleFlag Receive IDLE flag.
kUSART_TxReady Transmit ready flag.
kUSART_TxIdleFlag Transmit idle flag.
kUSART_CtsState Cts pin status.
kUSART_DeltaCtsFlag Cts pin change flag.
kUSART_TxDisableFlag Transmit disable flag.
kUSART_HardwareOverrunFlag Hardware over run flag.
kUSART_RxBreakFlag Receive break flag.
kUSART_RxStartFlag receive start flag.
kUSART_FramErrorFlag Frame error flag.
kUSART_ParityErrorFlag Parity error flag.
kUSART_RxNoiseFlag Receive noise flag.
kUSART_AutoBaudErrorFlag Auto baud error flag.

6.3.6 Function Documentation

6.3.6.1 uint32_t USART_GetInstance (USART_Type ∗ *base*)

6.3.6.2 status_t USART_Init (USART_Type ∗ *base*, const usart_config_t ∗ *config*, uint32_t *srcClock_Hz*)

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the USART_GetDefaultConfig() function. Example below shows how to use this API to configure USART.

```
* usart_config_t usartConfig;
* usartConfig.baudRate_Bps = 115200U;
* usartConfig.parityMode = kUSART_ParityDisabled;
* usartConfig.stopBitCount = kUSART_OneStopBit;
* USART_Init(USART1, &usartConfig, 20000000U);
*
```

Parameters

base	USART peripheral base address.
config	Pointer to user-defined configuration structure.
srcClock_Hz	USART clock source frequency in HZ.

Return values

kStatus_USART_-BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_InvalidArgument	USART base address is not valid
kStatus_Success	Status USART initialize succeed

6.3.6.3 void USART_Deinit (USART_Type ∗ *base*)

This function waits for TX complete, disables the USART clock.

Parameters

base	USART peripheral base address.

6.3.6.4 void USART_GetDefaultConfig (usart_config_t ∗ *config*)

This function initializes the USART configuration structure to a default value. The default values are-: usartConfig->baudRate_Bps = 9600U; usartConfig->parityMode = kUSART_ParityDisabled; usartConfig->stopBitCount = kUSART_OneStopBit; usartConfig->bitCountPerChar = kUSART_8BitsPerChar; usartConfig->loopback = false; usartConfig->enableTx = false; usartConfig->enableRx = false; ...

Parameters

config	Pointer to configuration structure.

6.3.6.5 status_t USART_SetBaudRate (USART_Type * *base*, uint32_t *baudrate_Bps*, uint32_t *srcClock_Hz*)

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the USART_Init.

```
*  USART_SetBaudRate(USART1, 115200U, 20000000U);
*
```

Parameters

base	USART peripheral base address.
baudrate_Bps	USART baudrate to be set.
srcClock_Hz	USART clock source frequency in HZ.

Return values

kStatus_USART_-BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_Success	Set baudrate succeed.
kStatus_InvalidArgument	One or more arguments are invalid.

6.3.6.6 static uint32_t USART_GetStatusFlags (USART_Type * *base*) [inline], [static]

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators _usart_flags. To check a specific status, compare the return value with enumerators in _usart_flags. For example, to check whether the RX is ready:

```
*     if (kUSART_RxReady & USART_GetStatusFlags(USART1))
*     {
*         ...
*     }
*
```

Parameters

base	USART peripheral base address.

Returns

USART status flags which are ORed by the enumerators in the _usart_flags.

6.3.6.7 static void USART_ClearStatusFlags (USART_Type * *base*, uint32_t *mask*) [inline], [static]

This function clear supported USART status flags For example:

```
*     USART_ClearStatusFlags(USART1,
      kUSART_HardwareOverrunFlag)
*
```

Parameters

base	USART peripheral base address.
mask	status flags to be cleared.

6.3.6.8 static void USART_EnableInterrupts (USART_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _usart_interrupt_enable. For example, to enable TX ready interrupt and RX ready interrupt:

```
*     USART_EnableInterrupts(USART1,
      kUSART_RxReadyInterruptEnable |
      kUSART_TxReadyInterruptEnable);
*
```

Parameters

base	USART peripheral base address.

mask	The interrupts to enable. Logical OR of _usart_interrupt_enable.

6.3.6.9 static void USART_DisableInterrupts (USART_Type ∗ *base*, uint32_t *mask*) [inline], [static]

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See _usart_interrupt_enable. This example shows how to disable the TX ready interrupt and RX ready interrupt:

```
*     USART_DisableInterrupts(USART1,
      kUSART_TxReadyInterruptEnable |
      kUSART_RxReadyInterruptEnable);
*
```

Parameters

base	USART peripheral base address.
mask	The interrupts to disable. Logical OR of _usart_interrupt_enable.

6.3.6.10 static uint32_t USART_GetEnabledInterrupts (USART_Type ∗ *base*) [inline], [static]

This function returns the enabled USART interrupts.

Parameters

base	USART peripheral base address.

6.3.6.11 static void USART_EnableContinuousSCLK (USART_Type ∗ *base*, bool *enable*) [inline], [static]

By default, SCLK is only output while data is being transmitted in synchronous mode. Enable this funciton, SCLK will run continuously in synchronous mode, allowing characters to be received on Un_RxD independently from transmission on Un_TXD).

Parameters

base	USART peripheral base address.
enable	Enable Continuous Clock generation mode or not, true for enable and false for disable.

6.3.6.12 static void USART_EnableAutoClearSCLK (USART_Type * *base,* bool *enable*) [inline], [static]

While enable this cuntion, the Continuous Clock bit is automatically cleared when a complete character has been received. This bit is cleared at the same time.

Parameters

base	USART peripheral base address.
enable	Enable auto clear or not, true for enable and false for disable.

6.3.6.13 static void USART_EnableCTS (USART_Type * *base,* bool *enable*) [inline], [static]

This function will determine whether CTS is used for flow control.

Parameters

base	USART peripheral base address.
enable	Enable CTS or not, true for enable and false for disable.

6.3.6.14 static void USART_EnableTx (USART_Type * *base,* bool *enable*) [inline], [static]

This function will enable or disable the USART transmit.

Parameters

base	USART peripheral base address.
enable	true for enable and false for disable.

6.3.6.15 static void USART_EnableRx (USART_Type * *base,* bool *enable*) [inline], [static]

This function will enable or disable the USART receive. Note: if the transmit is enabled, the receive will not be disabled.

Parameters

base	USART peripheral base address.
enable	true for enable and false for disable.

6.3.6.16 static void USART_WriteByte (USART_Type ∗ *base,* uint8_t *data*) [inline], [static]

This function will writes data to the TXDAT automatly.The upper layer must ensure that TXDATA has space for data to write before calling this function.

Parameters

base	USART peripheral base address.
data	The byte to write.

6.3.6.17 static uint8_t USART_ReadByte (USART_Type ∗ *base*) [inline], [static]

This function reads data from the RXDAT automatly. The upper layer must ensure that the RXDAT is not empty before calling this function.

Parameters

base	USART peripheral base address.

Returns

The byte read from USART data register.

6.3.6.18 void USART_WriteBlocking (USART_Type ∗ *base,* const uint8_t ∗ *data,* size_t *length*)

This function polls the TX register, waits for the TX register to be empty.

Parameters

base	USART peripheral base address.
data	Start address of the data to write.
length	Size of the data to write.

6.3.6.19 status_t USART_ReadBlocking (USART_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the RX register, waits for the RX register to be full.

Parameters

base	USART peripheral base address.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_USART_-FramingError	Receiver overrun happened while receiving data.
kStatus_USART_Parity-Error	Noise error happened while receiving data.
kStatus_USART_Noise-Error	Framing error happened while receiving data.
kStatus_USART_RxError	Overflow or underflow happened.
kStatus_Success	Successfully received all data.

6.3.6.20 status_t USART_TransferCreateHandle (USART_Type * *base*, usart_handle_t * *handle*, usart_transfer_callback_t *callback*, void * *userData*)

This function initializes the USART handle which can be used for other USART transactional APIs. Usually, for a specified USART instance, call this API once to get the initialized handle.

Parameters

base	USART peripheral base address.

handle	USART handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

6.3.6.21 status_t USART_TransferSendNonBlocking (USART_Type * *base*, usart_handle_t * *handle*, usart_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 IRQ handler, the USART driver calls the callback function and passes the kStatus_USART_TxIdle as status parameter.

Note

> The kStatus_USART_TxIdle is passed to the upper layer when all data is written to the TX register. However it does not ensure that all data are sent out. Before disabling the TX, check the kUSART-_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
xfer	USART transfer structure. See usart_transfer_t.

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_USART_TxBusy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	Invalid argument.

6.3.6.22 void USART_TransferStartRingBuffer (USART_Type * *base*, usart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

This function sets up the RX ring buffer to a specific USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the USART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if ringBuffer-Size is 32, then only 31 bytes are used for saving data.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
ringBuffer	Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
ringBufferSize	size of the ring buffer.

6.3.6.23 void USART_TransferStopRingBuffer (USART_Type * *base,* usart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

6.3.6.24 size_t USART_TransferGetRxRingBufferLength (usart_handle_t * *handle*)

Parameters

handle	USART handle pointer.

Returns

Length of received data in RX ring buffer.

6.3.6.25 void USART_TransferAbortSend (USART_Type * *base,* usart_handle_t * *handle*)

This function aborts the interrupt driven data sending. The user can get the remainBtyes to find out how many bytes are still not sent out.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

6.3.6.26 status_t USART_TransferGetSendCount (USART_Type * *base*, usart_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to USART TX register by interrupt method.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

6.3.6.27 status_t USART_TransferReceiveNonBlocking (USART_Type * *base*, usart_handle_t * *handle*, usart_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 USART driver. When the new data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter kStatus_USART_RxIdle. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the xfer->data and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the xfer->data[5]. When 5 bytes are received, the USART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the xfer->data. When all data is received, the upper layer is notified.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
xfer	USART transfer structure, see usart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into transmit queue.
kStatus_USART_RxBusy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

6.3.6.28 void USART_TransferAbortReceive (USART_Type ∗ *base*, usart_handle_t ∗ *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

6.3.6.29 status_t USART_TransferGetReceiveCount (USART_Type ∗ *base*, usart_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes that have been received.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

6.3.6.30 void USART_TransferHandleIRQ (USART_Type ∗ *base*, usart_handle_t ∗ *handle*)

This function handles the USART transmit and receive IRQ request.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

Chapter 7
IOCON: I/O pin configuration

7.1 Overview

The MCUXpresso SDK provides Peripheral driver for the I/O pin configuration (IOCON) module of M-CUXpresso SDK devices.

7.2 Function groups

7.2.1 Pin mux set

The function IOCONPinMuxSet() set pinmux for single pin according to selected configuration.

7.2.2 Pin mux set

The function IOCON_SetPinMuxing() set pinmux for group of pins according to selected configuration.

7.3 Typical use case

Example use of IOCON API to selection of GPIO mode. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/iocon

Files

- file fsl_iocon.h

Data Structures

- struct iocon_group_t
  Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format. More...

Functions

- __STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t ionumber, uint32_t modefunc)
  IOCON function and mode selection definitions.
- __STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base, const iocon_group_t *pin-Array, uint32_t arrayLength)
  Set all I/O Control pin muxing.

Driver version

- #define LPC_IOCON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  IOCON driver version 2.0.0.

7.4 Data Structure Documentation

7.4.1 struct iocon_group_t

7.5 Macro Definition Documentation

7.5.1 #define LPC_IOCON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

7.6 Function Documentation

7.6.1 __STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type ∗ *base,* uint8_t *ionumber,* uint32_t *modefunc*)

Note

See the User Manual for specific modes and functions supported by the various pins. Sets I/O Control pin mux

Parameters

base	: The base of IOCON peripheral on the chip
ionumber	: GPIO number to mux
modefunc	: OR'ed values of type IOCON_∗

Returns

Nothing

7.6.2 __STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type ∗ *base,* const iocon_group_t ∗ *pinArray,* uint32_t *arrayLength*)

Parameters

base	: The base of IOCON peripheral on the chip
pinArray	: Pointer to array of pin mux selections
arrayLength	: Number of entries in pinArray

Returns

Nothing

Chapter 8
CTIMER: Standard counter/timers

8.1 Overview

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

8.2 Function groups

The cTimer driver supports the generation of PWM signals, input capture, and setting up the timer match conditions.

8.2.1 Initialization and deinitialization

The function CTIMER_Init() initializes the cTimer with specified configurations. The function CTIMER_GetDefaultConfig() gets the default configurations. The initialization function configures the counter/timer mode and input selection when running in counter mode.

The function CTIMER_Deinit() stops the timer and turns off the module clock.

8.2.2 PWM Operations

The function CTIMER_SetupPwm() sets up channels for PWM output. Each channel has its own duty cycle, however the same PWM period 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 CTIMER_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular channel.

8.2.3 Match Operation

The function CTIMER_SetupMatch() sets up channels for match operation. Each channel is configured with a match value: if the counter should stop on match, if counter should reset on match, and output pin action. The output signal can be cleared, set, or toggled on match.

8.2.4 Input capture operations

The function CTIMER_SetupCapture() sets up an channel for input capture. The user can specify the capture edge and if a interrupt should be generated when processing the input signal.

8.3 Typical use case

8.3.1 Match example

Set up a match channel to toggle output when a match occurs. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ctimer

8.3.2 PWM output example

Set up a channel for PWM output. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ctimer

Files

- file fsl_ctimer.h

Data Structures

- struct ctimer_match_config_t
  Match configuration. More...
- struct ctimer_config_t
  Timer configuration structure. More...

Enumerations

- enum ctimer_capture_channel_t {
  kCTIMER_Capture_0 = 0U,
  kCTIMER_Capture_1,
  kCTIMER_Capture_2 }
  List of Timer capture channels.
- enum ctimer_capture_edge_t {
  kCTIMER_Capture_RiseEdge = 1U,
  kCTIMER_Capture_FallEdge = 2U,
  kCTIMER_Capture_BothEdge = 3U }
  List of capture edge options.
- enum ctimer_match_t {
  kCTIMER_Match_0 = 0U,
  kCTIMER_Match_1,
  kCTIMER_Match_2,
  kCTIMER_Match_3 }
  List of Timer match registers.
- enum ctimer_match_output_control_t {
  kCTIMER_Output_NoAction = 0U,
  kCTIMER_Output_Clear,
  kCTIMER_Output_Set,
  kCTIMER_Output_Toggle }

List of output control options.
- enum ctimer_timer_mode_t
  List of Timer modes.
- enum ctimer_interrupt_enable_t {
  kCTIMER_Match0InterruptEnable = CTIMER_MCR_MR0I_MASK,
  kCTIMER_Match1InterruptEnable = CTIMER_MCR_MR1I_MASK,
  kCTIMER_Match2InterruptEnable = CTIMER_MCR_MR2I_MASK,
  kCTIMER_Match3InterruptEnable = CTIMER_MCR_MR3I_MASK,
  kCTIMER_Capture0InterruptEnable = CTIMER_CCR_CAP0I_MASK,
  kCTIMER_Capture1InterruptEnable = CTIMER_CCR_CAP1I_MASK,
  kCTIMER_Capture2InterruptEnable = CTIMER_CCR_CAP2I_MASK }
  List of Timer interrupts.
- enum ctimer_status_flags_t {
  kCTIMER_Match0Flag = CTIMER_IR_MR0INT_MASK,
  kCTIMER_Match1Flag = CTIMER_IR_MR1INT_MASK,
  kCTIMER_Match2Flag = CTIMER_IR_MR2INT_MASK,
  kCTIMER_Match3Flag = CTIMER_IR_MR3INT_MASK,
  kCTIMER_Capture0Flag = CTIMER_IR_CR0INT_MASK,
  kCTIMER_Capture1Flag = CTIMER_IR_CR1INT_MASK,
  kCTIMER_Capture2Flag = CTIMER_IR_CR2INT_MASK }
  List of Timer flags.
- enum ctimer_callback_type_t {
  kCTIMER_SingleCallback,
  kCTIMER_MultipleCallback }
  Callback type when registering for a callback.

Functions

- void CTIMER_SetupMatch (CTIMER_Type *base, ctimer_match_t matchChannel, const ctimer_match_config_t *config)
  Setup the match register.
- void CTIMER_SetupCapture (CTIMER_Type *base, ctimer_capture_channel_t capture, ctimer_capture_edge_t edge, bool enableInt)
  Setup the capture.
- static uint32_t CTIMER_GetTimerCountValue (CTIMER_Type *base)
  Get the timer count value from TC register.
- void CTIMER_RegisterCallBack (CTIMER_Type *base, ctimer_callback_t *cb_func, ctimer_callback_type_t cb_type)
  Register callback.
- static void CTIMER_Reset (CTIMER_Type *base)
  Reset the counter.

Driver version

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

Initialization and deinitialization

- void CTIMER_Init (CTIMER_Type ∗base, const ctimer_config_t ∗config)
  Ungates the clock and configures the peripheral for basic operation.
- void CTIMER_Deinit (CTIMER_Type ∗base)
  Gates the timer clock.
- void CTIMER_GetDefaultConfig (ctimer_config_t ∗config)
  Fills in the timers configuration structure with the default settings.

PWM setup operations

- status_t CTIMER_SetupPwmPeriod (CTIMER_Type ∗base, ctimer_match_t matchChannel, uint32_t pwmPeriod, uint32_t pulsePeriod, bool enableInt)
  Configures the PWM signal parameters.
- status_t CTIMER_SetupPwm (CTIMER_Type ∗base, ctimer_match_t matchChannel, uint8_t dutyCyclePercent, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, bool enableInt)
  Configures the PWM signal parameters.
- static void CTIMER_UpdatePwmPulsePeriod (CTIMER_Type ∗base, ctimer_match_t matchChannel, uint32_t pulsePeriod)
  Updates the pulse period of an active PWM signal.
- void CTIMER_UpdatePwmDutycycle (CTIMER_Type ∗base, ctimer_match_t matchChannel, uint8_t dutyCyclePercent)
  Updates the duty cycle of an active PWM signal.

Interrupt Interface

- static void CTIMER_EnableInterrupts (CTIMER_Type ∗base, uint32_t mask)
  Enables the selected Timer interrupts.
- static void CTIMER_DisableInterrupts (CTIMER_Type ∗base, uint32_t mask)
  Disables the selected Timer interrupts.
- static uint32_t CTIMER_GetEnabledInterrupts (CTIMER_Type ∗base)
  Gets the enabled Timer interrupts.

Status Interface

- static uint32_t CTIMER_GetStatusFlags (CTIMER_Type ∗base)
  Gets the Timer status flags.
- static void CTIMER_ClearStatusFlags (CTIMER_Type ∗base, uint32_t mask)
  Clears the Timer status flags.

Counter Start and Stop

- static void CTIMER_StartTimer (CTIMER_Type ∗base)
  Starts the Timer counter.
- static void CTIMER_StopTimer (CTIMER_Type ∗base)
  Stops the Timer counter.

8.4 Data Structure Documentation

8.4.1 struct ctimer_match_config_t

This structure holds the configuration settings for each match register.

Data Fields

- uint32_t matchValue
  This is stored in the match register.
- bool enableCounterReset
  true: Match will reset the counter false: Match will not reser the counter
- bool enableCounterStop
  true: Match will stop the counter false: Match will not stop the counter
- ctimer_match_output_control_t outControl
  Action to be taken on a match on the EM bit/output.
- bool outPinInitState
  Initial value of the EM bit/output.
- bool enableInterrupt
  true: Generate interrupt upon match false: Do not generate interrupt on match

8.4.2 struct ctimer_config_t

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

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

Data Fields

- ctimer_timer_mode_t mode
  Timer mode.
- ctimer_capture_channel_t input
  Input channel to increment the timer, used only in timer modes that rely on this input signal to increment TC.
- uint32_t prescale
  Prescale value.

8.5 Enumeration Type Documentation

8.5.1 enum ctimer_capture_channel_t

Enumerator

kCTIMER_Capture_0 Timer capture channel 0.

kCTIMER_Capture_1 Timer capture channel 1.
kCTIMER_Capture_2 Timer capture channel 2.

8.5.2 enum ctimer_capture_edge_t

Enumerator

kCTIMER_Capture_RiseEdge Capture on rising edge.
kCTIMER_Capture_FallEdge Capture on falling edge.
kCTIMER_Capture_BothEdge Capture on rising and falling edge.

8.5.3 enum ctimer_match_t

Enumerator

kCTIMER_Match_0 Timer match register 0.
kCTIMER_Match_1 Timer match register 1.
kCTIMER_Match_2 Timer match register 2.
kCTIMER_Match_3 Timer match register 3.

8.5.4 enum ctimer_match_output_control_t

Enumerator

kCTIMER_Output_NoAction No action is taken.
kCTIMER_Output_Clear Clear the EM bit/output to 0.
kCTIMER_Output_Set Set the EM bit/output to 1.
kCTIMER_Output_Toggle Toggle the EM bit/output.

8.5.5 enum ctimer_interrupt_enable_t

Enumerator

kCTIMER_Match0InterruptEnable Match 0 interrupt.
kCTIMER_Match1InterruptEnable Match 1 interrupt.
kCTIMER_Match2InterruptEnable Match 2 interrupt.
kCTIMER_Match3InterruptEnable Match 3 interrupt.
kCTIMER_Capture0InterruptEnable Capture 0 interrupt.
kCTIMER_Capture1InterruptEnable Capture 1 interrupt.
kCTIMER_Capture2InterruptEnable Capture 2 interrupt.

8.5.6 enum ctimer_status_flags_t

Enumerator

kCTIMER_Match0Flag Match 0 interrupt flag.
kCTIMER_Match1Flag Match 1 interrupt flag.
kCTIMER_Match2Flag Match 2 interrupt flag.
kCTIMER_Match3Flag Match 3 interrupt flag.
kCTIMER_Capture0Flag Capture 0 interrupt flag.
kCTIMER_Capture1Flag Capture 1 interrupt flag.
kCTIMER_Capture2Flag Capture 2 interrupt flag.

8.5.7 enum ctimer_callback_type_t

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

Enumerator

kCTIMER_SingleCallback Single Callback type where there is only one callback for the timer. based on the status flags different channels needs to be handled differently
kCTIMER_MultipleCallback Multiple Callback type where there can be 8 valid callbacks, one per channel. for both match/capture

8.6 Function Documentation

8.6.1 void CTIMER_Init (CTIMER_Type * *base*, const ctimer_config_t * *config*)

Note

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

Parameters

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

8.6.2 void CTIMER_Deinit (CTIMER_Type * *base*)

Parameters

base	Ctimer peripheral base address

8.6.3 void CTIMER_GetDefaultConfig (ctimer_config_t * *config*)

The default values are:

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

Parameters

config	Pointer to the user configuration structure.

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

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function will assign match channel 3 to set the PWM cycle.

Note

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

Parameters

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

Returns

kStatus_Success on success kStatus_Fail If matchChannel passed in is 3; this channel is reserved to set the PWM period

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

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function will assign match channel 3 to set the PWM cycle.

Note

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

Parameters

base	Ctimer peripheral base address
matchChannel	Match pin to be used to output the PWM signal
dutyCycle-Percent	PWM pulse width; the value should be between 0 to 100
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	Timer counter clock in Hz
enableInt	Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt is generated

Returns

kStatus_Success on success kStatus_Fail If matchChannel passed in is 3; this channel is reserved to set the PWM cycle

8.6.6 static void CTIMER_UpdatePwmPulsePeriod (CTIMER_Type ∗ *base*, ctimer_match_t *matchChannel*, uint32_t *pulsePeriod*) [inline], [static]

Parameters

base	Ctimer peripheral base address

matchChannel	Match pin to be used to output the PWM signal
pulsePeriod	New PWM pulse width match value

8.6.7 void CTIMER_UpdatePwmDutycycle (CTIMER_Type ∗ *base*, ctimer_match_t *matchChannel*, uint8_t *dutyCyclePercent*)

Note

Please use CTIMER_UpdatePwmPulsePeriod to update the PWM with high resolution.

Parameters

base	Ctimer peripheral base address
matchChannel	Match pin to be used to output the PWM signal
dutyCycle-Percent	New PWM pulse width; the value should be between 0 to 100

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

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

Parameters

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

8.6.9 void CTIMER_SetupCapture (CTIMER_Type ∗ *base*, ctimer_capture-_channel_t *capture*, ctimer_capture_edge_t *edge*, bool *enableInt*)

Parameters

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

8.6.10 static uint32_t CTIMER_GetTimerCountValue (CTIMER_Type ∗ *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address.

Returns

return the timer count value.

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

Parameters

base	Ctimer peripheral base address
cb_func	callback function
cb_type	callback function type, singular or multiple

8.6.12 static void CTIMER_EnableInterrupts (CTIMER_Type ∗ *base,* uint32_t *mask*) [inline], [static]

Parameters

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

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

Parameters

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

8.6.14 static uint32_t CTIMER_GetEnabledInterrupts (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration ctimer_interrupt_-enable_t

8.6.15 static uint32_t CTIMER_GetStatusFlags (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

Returns

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

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

Parameters

base	Ctimer peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration ctimer-_status_flags_t

8.6.17 static void CTIMER_StartTimer (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

8.6.18 static void CTIMER_StopTimer (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

8.6.19 static void CTIMER_Reset (CTIMER_Type * *base*) [inline], [static]

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

Parameters

base	Ctimer peripheral base address

Chapter 9
CAPT: Capacitive Touch

9.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Capacitive Touch (CAPT) module of MCUXpresso SDK devices.

The Capacitive Touch module measures the change in capacitance of an electrode plate when an earth-ground connected object (for example, the finger or stylus) is brought within close proximity. Simply stated, the module delivers a small charge to an X capacitor (a mutual capacitance touch sensor), then transfers that charge to a larger Y capacitor (the measurement capacitor), and counts the number of iterations necessary for the voltage across the Y capacitor to cross a predetermined threshold.

9.2 Typical use case

9.2.1 Normal Configuration

See the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/capt/capt_basic.

Files

- file fsl_capt.h

Data Structures

- struct capt_config_t
  The structure for CAPT basic configuration. More...
- struct capt_touch_data_t
  The structure for storing touch data. More...

Enumerations

- enum _capt_xpins {
  kCAPT_X0Pin = 1U << 0U,
  kCAPT_X1Pin = 1U << 1U,
  kCAPT_X2Pin = 1U << 2U,
  kCAPT_X3Pin = 1U << 3U,
  kCAPT_X4Pin = 1U << 4U,
  kCAPT_X5Pin = 1U << 5U,
  kCAPT_X6Pin = 1U << 6U,
  kCAPT_X7Pin = 1U << 7U,
  kCAPT_X8Pin = 1U << 8U,
  kCAPT_X9Pin = 1U << 9U,
  kCAPT_X10Pin = 1U << 10U,
  kCAPT_X11Pin = 1U << 11U,
  kCAPT_X12Pin = 1U << 12U,
  kCAPT_X13Pin = 1U << 13U,
  kCAPT_X14Pin = 1U << 14U,
  kCAPT_X15Pin = 1U << 15U }
  The enumeration for X pins.
- enum _capt_interrupt_enable {
  kCAPT_InterruptOfYesTouchEnable,
  kCAPT_InterruptOfNoTouchEnable,
  kCAPT_InterruptOfPollDoneEnable = CAPT_INTENSET_POLLDONE_MASK,
  kCAPT_InterruptOfTimeOutEnable = CAPT_INTENSET_TIMEOUT_MASK,
  kCAPT_InterruptOfOverRunEnable = CAPT_INTENSET_OVERUN_MASK }
  The enumeration for enabling/disabling interrupts.
- enum _capt_interrupt_status_flags {
  kCAPT_InterruptOfYesTouchStatusFlag = CAPT_INTSTAT_YESTOUCH_MASK,
  kCAPT_InterruptOfNoTouchStatusFlag = CAPT_INTSTAT_NOTOUCH_MASK,
  kCAPT_InterruptOfPollDoneStatusFlag = CAPT_INTSTAT_POLLDONE_MASK,
  kCAPT_InterruptOfTimeOutStatusFlag = CAPT_INTSTAT_TIMEOUT_MASK,
  kCAPT_InterruptOfOverRunStatusFlag = CAPT_INTSTAT_OVERUN_MASK }
  The enumeration for interrupt status flags.
- enum _capt_status_flags {
  kCAPT_BusyStatusFlag = CAPT_STATUS_BUSY_MASK,
  kCAPT_XMAXStatusFlag = CAPT_STATUS_XMAX_MASK }
  The enumeration for CAPT status flags.
- enum capt_trigger_mode_t {
  kCAPT_YHPortTriggerMode = 0U,
  kCAPT_ComparatorTriggerMode = 1U }
  The enumeration for CAPT trigger mode.
- enum capt_inactive_xpins_mode_t {
  kCAPT_InactiveXpinsHighZMode,
  kCAPT_InactiveXpinsDrivenLowMode }
  The enumeration for the inactive X pins mode.
- enum capt_measurement_delay_t {

kCAPT_MeasureDelayNoWait = 0U,
kCAPT_MeasureDelayWait3FCLKs = 1U,
kCAPT_MeasureDelayWait5FCLKs = 2U,
kCAPT_MeasureDelayWait9FCLKs = 3U }
The enumeration for the delay of measuring voltage state.

- enum capt_reset_delay_t {
kCAPT_ResetDelayNoWait = 0U,
kCAPT_ResetDelayWait3FCLKs = 1U,
kCAPT_ResetDelayWait5FCLKs = 2U,
kCAPT_ResetDelayWait9FCLKs = 3U }
The enumeration for the delay of reseting or draining Cap.
- enum capt_polling_mode_t {
kCAPT_PollInactiveMode,
kCAPT_PollNowMode = 1U,
kCAPT_PollContinuousMode }
The enumeration of CAPT polling mode.
- enum capt_dma_mode_t {
kCAPT_DMATriggerOnTouchMode = 1U,
kCAPT_DMATriggerOnBothMode = 2U,
kCAPT_DMATriggerOnAllMode = 3U }
The enumeration of CAPT DMA trigger mode.

Variables

- bool capt_config_t::enableWaitMode
If enable the wait mode, when the touch event occurs, the module will wait until the TOUCH register is read before starting the next measurement.
- bool capt_config_t::enableTouchLower
enableTouchLower = true: Trigger at count < TCNT is a touch.
- uint8_t capt_config_t::clockDivider
Function clock divider.
- uint8_t capt_config_t::timeOutCount
Sets the count value at which a time-out event occurs if a measurement has not triggered.
- uint8_t capt_config_t::pollCount
Sets the time delay between polling rounds (successive sets of X measurements).
- uint16_t capt_config_t::enableXpins
Selects which of the available X pins are enabled.
- capt_trigger_mode_t capt_config_t::triggerMode
Select the menthods of measuring the voltage across the measurement capacitor.
- capt_inactive_xpins_mode_t capt_config_t::XpinsMode
Determines how X pins enabled in the XPINSEL field are controlled when not active.
- capt_measurement_delay_t capt_config_t::mDelay
`Set the time delay after entering step 3 (measure voltage state), before` *sampling the YH port pin or analog comarator output.*
- capt_reset_delay_t capt_config_t::rDelay
Set the number of divided FCLKs the module will remain in Reset or Draining Cap.
- bool capt_touch_data_t::yesTimeOut
'true': if the measurement resulted in a time-out event, 'false': otherwise.
- bool capt_touch_data_t::yesTouch

'true': if the trigger is due to a touch even, 'false': if the trigger is due to a no-touch event.

- uint8_t capt_touch_data_t::XpinsIndex
  Contains the index of the X pin for the current measurement, or lowest X for a multiple-pin poll now measurement.
- uint8_t capt_touch_data_t::sequenceNumber
  Contains the 4-bit(0-7) sequence number, which increments at the end of each polling round.
- uint16_t capt_touch_data_t::count
  Contains the count value reached at trigger or time-out.

Driver version

- #define FSL_CAPT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  CAPT driver version 2.0.1.

Initialization

- void CAPT_Init (CAPT_Type ∗base, const capt_config_t ∗config)
  Initialize the CAPT module.
- void CAPT_Deinit (CAPT_Type ∗base)
  De-initialize the CAPT module.
- void CAPT_GetDefaultConfig (capt_config_t ∗config)
  Gets an available pre-defined settings for the CAPT's configuration.
- static void CAPT_SetThreshold (CAPT_Type ∗base, uint32_t count)
  Set Sets the count threshold in divided FCLKs between touch and no-touch.
- void CAPT_SetPollMode (CAPT_Type ∗base, capt_polling_mode_t mode)
  Set the CAPT polling mode.
- static void CAPT_EnableInterrupts (CAPT_Type ∗base, uint32_t mask)
  Enable interrupt features.
- static void CAPT_DisableInterrupts (CAPT_Type ∗base, uint32_t mask)
  Disable interrupt features.
- static uint32_t CAPT_GetInterruptStatusFlags (CAPT_Type ∗base)
  Get CAPT interrupts' status flags.
- static void CAPT_ClearInterruptStatusFlags (CAPT_Type ∗base, uint32_t mask)
  Clear the interrupts' status flags.
- static uint32_t CAPT_GetStatusFlags (CAPT_Type ∗base)
  Get CAPT status flags.
- bool CAPT_GetTouchData (CAPT_Type ∗base, capt_touch_data_t ∗data)
  Get CAPT touch data.

9.3 Data Structure Documentation

9.3.1 struct capt_config_t

Data Fields

- bool enableWaitMode
  If enable the wait mode, when the touch event occurs, the module will wait until the TOUCH register is read before starting the next measurement.
- bool enableTouchLower
  enableTouchLower = true: Trigger at count < TCNT is a touch.

- uint8_t clockDivider
  Function clock divider.
- uint8_t timeOutCount
  Sets the count value at which a time-out event occurs if a measurement has not triggered.
- uint8_t pollCount
  Sets the time delay between polling rounds (successive sets of X measurements).
- uint16_t enableXpins
  Selects which of the available X pins are enabled.
- capt_trigger_mode_t triggerMode
  Select the menthods of measuring the voltage across the measurement capacitor.
- capt_inactive_xpins_mode_t XpinsMode
  Determines how X pins enabled in the XPINSEL field are controlled when not active.
- capt_measurement_delay_t mDelay
  `Set the time delay after entering step 3 (measure voltage state), before`
  sampling the YH port pin or analog comarator output.
- capt_reset_delay_t rDelay
  Set the number of divided FCLKs the module will remain in Reset or Draining Cap.

9.3.2 struct capt_touch_data_t

Data Fields

- bool yesTimeOut
  'true': if the measurement resulted in a time-out event, 'false': otherwise.
- bool yesTouch
  'true': if the trigger is due to a touch even, 'false': if the trigger is due to a no-touch event.
- uint8_t XpinsIndex
  Contains the index of the X pin for the current measurement, or lowest X for a multiple-pin poll now measurement.
- uint8_t sequenceNumber
  Contains the 4-bit(0-7) sequence number, which increments at the end of each polling round.
- uint16_t count
  Contains the count value reached at trigger or time-out.

9.4 Macro Definition Documentation

9.4.1 #define FSL_CAPT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

9.5 Enumeration Type Documentation

9.5.1 enum _capt_xpins

Enumerator

kCAPT_X0Pin CAPT_X0 pin.
kCAPT_X1Pin CAPT_X1 pin.
kCAPT_X2Pin CAPT_X2 pin.
kCAPT_X3Pin CAPT_X3 pin.

kCAPT_X4Pin CAPT_X4 pin.
kCAPT_X5Pin CAPT_X5 pin.
kCAPT_X6Pin CAPT_X6 pin.
kCAPT_X7Pin CAPT_X7 pin.
kCAPT_X8Pin CAPT_X8 pin.
kCAPT_X9Pin CAPT_X9 pin.
kCAPT_X10Pin CAPT_X10 pin.
kCAPT_X11Pin CAPT_X11 pin.
kCAPT_X12Pin CAPT_X12 pin.
kCAPT_X13Pin CAPT_X13 pin.
kCAPT_X14Pin CAPT_X14 pin.
kCAPT_X15Pin CAPT_X15 pin.

9.5.2 enum _capt_interrupt_enable

Enumerator

kCAPT_InterruptOfYesTouchEnable Generate interrupt when a touch has been detected.
kCAPT_InterruptOfNoTouchEnable Generate interrupt when a no-touch has been detected.
kCAPT_InterruptOfPollDoneEnable Genarate interrupt at the end of a polling round, or when a POLLNOW completes.
kCAPT_InterruptOfTimeOutEnable Generate interrupt when the count reaches the time-out count value before a trigger occurs.
kCAPT_InterruptOfOverRunEnable Generate interrupt when the Touch Data register has been updated before software has read the previous data, and the touch has been detected.

9.5.3 enum _capt_interrupt_status_flags

Enumerator

kCAPT_InterruptOfYesTouchStatusFlag YESTOUCH interrupt status flag.
kCAPT_InterruptOfNoTouchStatusFlag NOTOUCH interrupt status flag.
kCAPT_InterruptOfPollDoneStatusFlag POLLDONE interrupt status flag.
kCAPT_InterruptOfTimeOutStatusFlag TIMEOUT interrupt status flag.
kCAPT_InterruptOfOverRunStatusFlag OVERRUN interrupt status flag.

9.5.4 enum _capt_status_flags

Enumerator

kCAPT_BusyStatusFlag Set while a poll is currently in progress, otherwise cleared.

kCAPT_XMAXStatusFlag The maximum number of X pins available for a given device is equal to XMAX+1.

9.5.5 enum capt_trigger_mode_t

Enumerator

kCAPT_YHPortTriggerMode YH port pin trigger mode.
kCAPT_ComparatorTriggerMode Analog comparator trigger mode.

9.5.6 enum capt_inactive_xpins_mode_t

Enumerator

kCAPT_InactiveXpinsHighZMode Xpins enabled in the XPINSEL field are controlled to HIGH-Z mode when not active.
kCAPT_InactiveXpinsDrivenLowMode Xpins enabled in the XPINSEL field are controlled to be driven low mode when not active.

9.5.7 enum capt_measurement_delay_t

Enumerator

kCAPT_MeasureDelayNoWait Don't wait.
kCAPT_MeasureDelayWait3FCLKs Wait 3 divided FCLKs.
kCAPT_MeasureDelayWait5FCLKs Wait 5 divided FCLKs.
kCAPT_MeasureDelayWait9FCLKs Wait 9 divided FCLKs.

9.5.8 enum capt_reset_delay_t

Enumerator

kCAPT_ResetDelayNoWait Don't wait.
kCAPT_ResetDelayWait3FCLKs Wait 3 divided FCLKs.
kCAPT_ResetDelayWait5FCLKs Wait 5 divided FCLKs.
kCAPT_ResetDelayWait9FCLKs Wait 9 divided FCLKs.

9.5.9 enum capt_polling_mode_t

Enumerator

kCAPT_PollInactiveMode No measurements are taken, no polls are performed. The module remains in the Reset Cap.

kCAPT_PollNowMode Immediately launches (ignoring Poll Delay) a one-time-only, simultaneous poll of all X pins that are enabled in the XPINSEL field of the Control register, then stops, returning to Reset/Draining Cap.

kCAPT_PollContinuousMode Polling rounds are continuously performed, by walking through the enabled X pins.

9.5.10 enum capt_dma_mode_t

Enumerator

kCAPT_DMATriggerOnTouchMode Trigger on touch.

kCAPT_DMATriggerOnBothMode Trigger on both touch and no-touch.

kCAPT_DMATriggerOnAllMode Trigger on all touch, no-touch and time-out.

9.6 Function Documentation

9.6.1 void CAPT_Init (CAPT_Type ∗ *base,* const capt_config_t ∗ *config*)

Parameters

base	CAPT peripheral base address.
config	Pointer to "capt_config_t" structure.

9.6.2 void CAPT_Deinit (CAPT_Type ∗ *base*)

Parameters

base	CAPT peripheral base address.

9.6.3 void CAPT_GetDefaultConfig (capt_config_t ∗ *config*)

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

```
*   config->enableWaitMode = false;
*   config->enableTouchLower = true;
*   config->clockDivider = 15U;
*   config->timeOutCount = 12U;
*   config->pollCount = 0U;
*   config->enableXpins = 0U;
*   config->triggerMode = kCAPT_YHPortTriggerMode;
*   config->XpinsMode = kCAPT_InactiveXpinsDrivenLowMode;
*   config->mDelay = kCAPT_MeasureDelayNoWait;
*   config->rDelay = kCAPT_ResetDelayWait9FCLKs;
*
```

Parameters

base	CAPT peripheral base address.
config	Pointer to the configuration structure.

9.6.4 static void CAPT_SetThreshold (CAPT_Type * *base*, uint32_t *count*) [inline], [static]

Parameters

base	CAPT peripheral base address.
count	The count threshold.

9.6.5 void CAPT_SetPollMode (CAPT_Type * *base*, capt_polling_mode_t *mode*)

Parameters

base	CAPT peripheral base address.
mode	The selection of polling mode.

9.6.6 static void CAPT_EnableInterrupts (CAPT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	CAPT peripheral base address.
mask	The mask of enabling interrupt features. Please refer to "_capt_interrupt_enable".

9.6.7 static void CAPT_DisableInterrupts (CAPT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	CAPT peripheral base address.
mask	The mask of disabling interrupt features. Please refer to "_capt_interrupt_enable".

9.6.8 static uint32_t CAPT_GetInterruptStatusFlags (CAPT_Type * *base*) [inline], [static]

Parameters

base	CAPT peripheral base address.

Returns

The mask of interrupts' status flags. please refer to "_capt_interrupt_status_flags".

9.6.9 static void CAPT_ClearInterruptStatusFlags (CAPT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	CAPT peripheral base address.
mask	The mask of clearing the interrupts' status flags, please refer to "_capt_interrupt_-status_flags".

9.6.10 static uint32_t CAPT_GetStatusFlags (CAPT_Type * *base*) [inline], [static]

Parameters

base	CAPT peripheral base address.

Returns

The mask of CAPT status flags. Please refer to "_capt_status_flags" Or use CAPT_GET_XMAX_-NUMBER(mask) to get XMAX number.

9.6.11 bool CAPT_GetTouchData (CAPT_Type ∗ *base*, capt_touch_data_t ∗ *data*)

Parameters

base	CAPT peripheral base address.
data	The structure to store touch data.

Returns

If return 'true', which means get valid data. if return 'false', which means get invalid data.

9.7 Variable Documentation

9.7.1 bool capt_config_t::enableWaitMode

Other-wise, measurements continue.

9.7.2 bool capt_config_t::enableTouchLower

Trigger at count > TCNT is a no-touch. enableTouchLower = false: Trigger at count > TCNT is a touch. Trigger at count < TCNT is a no-touch. Notice: TCNT will be set by "CAPT_DoCalibration" API.

9.7.3 uint8_t capt_config_t::clockDivider

The function clock is divided by clockDivider+1 to produce the divided FCLK for the module. The available range is 0-15.

9.7.4 uint8_t capt_config_t::timeOutCount

The time-out count value is calculated as $2^{\wedge}$timeOutCount. The available range is 0-12.

9.7.5 uint8_t capt_config_t::pollCount

After each polling round completes, the module will wait 4096 x PollCount divided FCLKs before starting the next polling round. The available range is 0-255.

9.7.6 uint16_t capt_config_t::enableXpins

Please refer to '_capt_xpins'. For example, if want to enable X0, X2 and X3 pins, you can set "enable-Xpins = kCAPT_X0Pin | kCAPT_X2Pin | kCAPT_X3Pin".

9.7.7 capt_trigger_mode_t capt_config_t::triggerMode

9.7.8 capt_inactive_xpins_mode_t capt_config_t::XpinsMode

9.7.9 capt_measurement_delay_t capt_config_t::mDelay

9.7.10 capt_reset_delay_t capt_config_t::rDelay

9.7.11 bool capt_touch_data_t::yesTimeOut

9.7.12 bool capt_touch_data_t::yesTouch

9.7.13 uint8_t capt_touch_data_t::XpinsIndex

9.7.14 uint8_t capt_touch_data_t::sequenceNumber

9.7.15 uint16_t capt_touch_data_t::count

Chapter 10
CRC: Cyclic Redundancy Check Driver

10.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Cyclic Redundancy Check (CRC) module of MCUXpresso SDK devices.

The cyclic redundancy check (CRC) module generates 16/32-bit CRC code for error detection. The CRC module also provides a programmable polynomial, seed, and other parameters required to implement a 16-bit or 32-bit CRC standard.

10.2 CRC Driver Initialization and Configuration

CRC_Init() function enables the clock gate for the CRC module in the SIM module and fully (re-)configures the CRC module according to the configuration structure. The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting a new checksum computation, the seed is set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed is set to the intermediate checksum value as obtained from previous calls to CRC_Get16bitResult() or CRC_Get32bitResult() function. After calling the CRC_Init(), one or multiple CRC_WriteData() calls follow to update the checksum with data and CRC_Get16bitResult() or CRC_Get32bitResult() follow to read the result. The crcResult member of the configuration structure determines whether the CRC_Get16bitResult() or CRC_Get32bitResult() return value is a final checksum or an intermediate checksum. The CRC_Init() function can be called as many times as required allowing for runtime changes of the CRC protocol.

CRC_GetDefaultConfig() function can be used to set the module configuration structure with parameters for CRC-16/CCIT-FALSE protocol.

10.3 CRC Write Data

The CRC_WriteData() function adds data to the CRC. Internally, it tries to use 32-bit reads and writes for all aligned data in the user buffer and 8-bit reads and writes for all unaligned data in the user buffer. This function can update the CRC with user-supplied data chunks of an arbitrary size, so one can update the CRC byte by byte or with all bytes at once. Prior to calling the CRC configuration function CRC_Init() fully specifies the CRC module configuration for the CRC_WriteData() call.

10.4 CRC Get Checksum

The CRC_Get16bitResult() or CRC_Get32bitResult() function reads the CRC module data register. Depending on the prior CRC module usage, the return value is either an intermediate checksum or the final checksum. For example, for 16-bit CRCs the following call sequences can be used.

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() to get the final checksum.

CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_Get16bitResult() to get the final checksum.

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() to get an intermediate checksum.

CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_Get16bitResult() to get an intermediate checksum.

10.5 Comments about API usage in RTOS

If multiple RTOS tasks share the CRC module to compute checksums with different data and/or protocols, the following needs to be implemented by the user.

The triplets

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() or CRC_Get32bitResult()

The triplets are protected by the RTOS mutex to protect the CRC module against concurrent accesses from different tasks. This is an example. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crc

Data Structures

- struct crc_config_t
  CRC protocol configuration. More...

Macros

- #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1
  Default configuration structure filled by CRC_GetDefaultConfig().

Enumerations

- enum crc_bits_t {
  kCrcBits16 = 0U,
  kCrcBits32 = 1U }
  CRC bit width.
- enum crc_result_t {
  kCrcFinalChecksum = 0U,
  kCrcIntermediateChecksum = 1U }
  CRC result type.

Functions

- void CRC_Init (CRC_Type ∗base, const crc_config_t ∗config)
  Enables and configures the CRC peripheral module.
- static void CRC_Deinit (CRC_Type ∗base)
  Disables the CRC peripheral module.
- void CRC_GetDefaultConfig (crc_config_t ∗config)

Loads default values to the CRC protocol configuration structure.
- void CRC_WriteData (CRC_Type ∗base, const uint8_t ∗data, size_t dataSize)
  Writes data to the CRC module.
- uint32_t CRC_Get32bitResult (CRC_Type ∗base)
  Reads the 32-bit checksum from the CRC module.
- uint16_t CRC_Get16bitResult (CRC_Type ∗base)
  Reads a 16-bit checksum from the CRC module.

Driver version

- #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  CRC driver version.

10.6 Data Structure Documentation

10.6.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

- uint32_t polynomial
  CRC Polynomial, MSBit first.
- uint32_t seed
  Starting checksum value.
- bool reflectIn
  Reflect bits on input.
- bool reflectOut
  Reflect bits on output.
- bool complementChecksum
  True if the result shall be complement of the actual checksum.
- crc_bits_t crcBits
  Selects 16- or 32- bit CRC protocol.
- crc_result_t crcResult
  Selects final or intermediate checksum return from CRC_Get16bitResult() or CRC_Get32bitResult()

10.6.1.0.0.4 Field Documentation

10.6.1.0.0.4.1 uint32_t crc_config_t::polynomial

Example polynomial: 0x1021 = 1_0000_0010_0001 = x^12+x^5+1

10.6.1.0.0.4.2 bool crc_config_t::reflectIn

10.6.1.0.0.4.3 bool crc_config_t::reflectOut

10.6.1.0.0.4.4 bool crc_config_t::complementChecksum

10.6.1.0.0.4.5 crc_bits_t crc_config_t::crcBits

10.7 Macro Definition Documentation

10.7.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Version 2.0.1.

Current version: 2.0.1

Change log:

- Version 2.0.1
  - move DATA and DATALL macro definition from header file to source file

10.7.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as defeault.

10.8 Enumeration Type Documentation

10.8.1 enum crc_bits_t

Enumerator

kCrcBits16 Generate 16-bit CRC code.
kCrcBits32 Generate 32-bit CRC code.

10.8.2 enum crc_result_t

Enumerator

kCrcFinalChecksum CRC data register read value is the final checksum. Reflect out and final xor protocol features are applied.
kCrcIntermediateChecksum CRC data register read value is intermediate checksum (raw value). Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for CRC_Init() to continue adding data to this checksum.

10.9 Function Documentation

10.9.1 void CRC_Init (CRC_Type ∗ *base,* const crc_config_t ∗ *config*)

This function enables the clock gate in the SIM module for the CRC peripheral. It also configures the CRC module and starts a checksum computation by writing the seed.

Parameters

base	CRC peripheral address.
config	CRC module configuration structure.

10.9.2 static void CRC_Deinit (CRC_Type ∗ *base*) [inline], [static]

This function disables the clock gate in the SIM module for the CRC peripheral.

Parameters

base	CRC peripheral address.

10.9.3 void CRC_GetDefaultConfig (crc_config_t ∗ *config*)

Loads default values to the CRC protocol configuration structure. The default values are as follows.

```
*   config->polynomial = 0x1021;
*   config->seed = 0xFFFF;
*   config->reflectIn = false;
*   config->reflectOut = false;
*   config->complementChecksum = false;
*   config->crcBits = kCrcBits16;
*   config->crcResult = kCrcFinalChecksum;
*
```

Parameters

config	CRC protocol configuration structure.

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

Writes input data buffer bytes to the CRC data register. The configured type of transpose is applied.

Parameters

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

10.9.5 uint32_t CRC_Get32bitResult (CRC_Type * *base*)

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

base	CRC peripheral address.

Returns

An intermediate or the final 32-bit checksum, after configured transpose and complement operations.

10.9.6 uint16_t CRC_Get16bitResult (CRC_Type * *base*)

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

base	CRC peripheral address.

Returns

An intermediate or the final 16-bit checksum, after configured transpose and complement operations.

Chapter 11
ADC: 12-bit SAR Analog-to-Digital Converter Driver

11.1 Overview

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

11.2 Typical use case

11.2.1 Polling Configuration

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

11.2.2 Interrupt Configuration

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

Files

- file fsl_adc.h

Data Structures

- struct adc_config_t
  Define structure for configuring the block. More...
- struct adc_conv_seq_config_t
  Define structure for configuring conversion sequence. More...
- struct adc_result_info_t
  Define structure of keeping conversion result information. More...

Enumerations

- enum _adc_status_flags {
  kADC_ThresholdCompareFlagOnChn0 = 1U << 0U,
  kADC_ThresholdCompareFlagOnChn1 = 1U << 1U,
  kADC_ThresholdCompareFlagOnChn2 = 1U << 2U,
  kADC_ThresholdCompareFlagOnChn3 = 1U << 3U,
  kADC_ThresholdCompareFlagOnChn4 = 1U << 4U,
  kADC_ThresholdCompareFlagOnChn5 = 1U << 5U,
  kADC_ThresholdCompareFlagOnChn6 = 1U << 6U,
  kADC_ThresholdCompareFlagOnChn7 = 1U << 7U,
  kADC_ThresholdCompareFlagOnChn8 = 1U << 8U,
  kADC_ThresholdCompareFlagOnChn9 = 1U << 9U,
  kADC_ThresholdCompareFlagOnChn10 = 1U << 10U,
  kADC_ThresholdCompareFlagOnChn11 = 1U << 11U,
  kADC_OverrunFlagForChn0,
  kADC_OverrunFlagForChn1,
  kADC_OverrunFlagForChn2,
  kADC_OverrunFlagForChn3,
  kADC_OverrunFlagForChn4,
  kADC_OverrunFlagForChn5,
  kADC_OverrunFlagForChn6,
  kADC_OverrunFlagForChn7,
  kADC_OverrunFlagForChn8,
  kADC_OverrunFlagForChn9,
  kADC_OverrunFlagForChn10,
  kADC_OverrunFlagForChn11,
  kADC_GlobalOverrunFlagForSeqA = 1U << 24U,
  kADC_GlobalOverrunFlagForSeqB = 1U << 25U,
  kADC_ConvSeqAInterruptFlag = 1U << 28U,
  kADC_ConvSeqBInterruptFlag = 1U << 29U,
  kADC_ThresholdCompareInterruptFlag = 1U << 30U,
  kADC_OverrunInterruptFlag = (int)(1U << 31U) }

  Flags.
- enum _adc_interrupt_enable {
  kADC_ConvSeqAInterruptEnable = ADC_INTEN_SEQA_INTEN_MASK,
  kADC_ConvSeqBInterruptEnable = ADC_INTEN_SEQB_INTEN_MASK,
  kADC_OverrunInterruptEnable = ADC_INTEN_OVR_INTEN_MASK }

  Interrupts.
- enum adc_clock_mode_t {
  kADC_ClockSynchronousMode,
  kADC_ClockAsynchronousMode = 1U }

  Define selection of clock mode.
- enum adc_vdda_range_t

  Definfe range of the analog supply voltage VDDA.
- enum adc_trigger_polarity_t {

kADC_TriggerPolarityNegativeEdge = 0U,
kADC_TriggerPolarityPositiveEdge = 1U }
Define selection of polarity of selected input trigger for conversion sequence.
- enum adc_priority_t {
kADC_PriorityLow = 0U,
kADC_PriorityHigh = 1U }
Define selection of conversion sequence's priority.
- enum adc_seq_interrupt_mode_t {
kADC_InterruptForEachConversion = 0U,
kADC_InterruptForEachSequence = 1U }
Define selection of conversion sequence's interrupt.
- enum adc_threshold_compare_status_t {
kADC_ThresholdCompareInRange = 0U,
kADC_ThresholdCompareBelowRange = 1U,
kADC_ThresholdCompareAboveRange = 2U }
Define status of threshold compare result.
- enum adc_threshold_crossing_status_t {
kADC_ThresholdCrossingNoDetected = 0U,
kADC_ThresholdCrossingDownward = 2U,
kADC_ThresholdCrossingUpward = 3U }
Define status of threshold crossing detection result.
- enum adc_threshold_interrupt_mode_t {
kADC_ThresholdInterruptDisabled = 0U,
kADC_ThresholdInterruptOnOutside = 1U,
kADC_ThresholdInterruptOnCrossing = 2U }
Define interrupt mode for threshold compare event.
- enum adc_inforesult_t {
kADC_Resolution12bitInfoResultShift = 0U,
kADC_Resolution10bitInfoResultShift = 2U,
kADC_Resolution8bitInfoResultShift = 4U,
kADC_Resolution6bitInfoResultShift = 6U }
Define the info result mode of different resolution.
- enum adc_tempsensor_common_mode_t {
kADC_HighNegativeOffsetAdded = 0x0U,
kADC_IntermediateNegativeOffsetAdded,
kADC_NoOffsetAdded = 0x8U,
kADC_LowPositiveOffsetAdded = 0xcU }
Define common modes for Temerature sensor.
- enum adc_second_control_t {
kADC_Impedance621Ohm = 0x1U << 9U,
kADC_Impedance55kOhm,
kADC_Impedance87kOhm = 0x1fU << 9U,
kADC_NormalFunctionalMode = 0x0U << 14U,
kADC_MultiplexeTestMode = 0x1U << 14U,
kADC_ADCInUnityGainMode = 0x2U << 14U }
Define source impedance modes for GPADC control.

Driver version

- #define FSL_ADC_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))
  ADC driver version 2.3.1.

Initialization and Deinitialization

- void ADC_Init (ADC_Type ∗base, const adc_config_t ∗config)
  Initialize the ADC module.
- void ADC_Deinit (ADC_Type ∗base)
  Deinitialize the ADC module.
- void ADC_GetDefaultConfig (adc_config_t ∗config)
  Gets an available pre-defined settings for initial configuration.
- bool ADC_DoSelfCalibration (ADC_Type ∗base, uint32_t frequency)
  Do the self calibration.

Control conversion sequence A.

- static void ADC_EnableConvSeqA (ADC_Type ∗base, bool enable)
  Enable the conversion sequence A.
- void ADC_SetConvSeqAConfig (ADC_Type ∗base, const adc_conv_seq_config_t ∗config)
  Configure the conversion sequence A.
- static void ADC_DoSoftwareTriggerConvSeqA (ADC_Type ∗base)
  Do trigger the sequence's conversion by software.
- static void ADC_EnableConvSeqABurstMode (ADC_Type ∗base, bool enable)
  Enable the burst conversion of sequence A.
- static void ADC_SetConvSeqAHighPriority (ADC_Type ∗base)
  Set the high priority for conversion sequence A.

Control conversion sequence B.

- static void ADC_EnableConvSeqB (ADC_Type ∗base, bool enable)
  Enable the conversion sequence B.
- void ADC_SetConvSeqBConfig (ADC_Type ∗base, const adc_conv_seq_config_t ∗config)
  Configure the conversion sequence B.
- static void ADC_DoSoftwareTriggerConvSeqB (ADC_Type ∗base)
  Do trigger the sequence's conversion by software.
- static void ADC_EnableConvSeqBBurstMode (ADC_Type ∗base, bool enable)
  Enable the burst conversion of sequence B.
- static void ADC_SetConvSeqBHighPriority (ADC_Type ∗base)
  Set the high priority for conversion sequence B.

Data result.

- bool ADC_GetConvSeqAGlobalConversionResult (ADC_Type ∗base, adc_result_info_t ∗info)
  Get the global ADC conversion infomation of sequence A.
- bool ADC_GetConvSeqBGlobalConversionResult (ADC_Type ∗base, adc_result_info_t ∗info)
  Get the global ADC conversion infomation of sequence B.
- bool ADC_GetChannelConversionResult (ADC_Type ∗base, uint32_t channel, adc_result_info_t ∗info)
  Get the channel's ADC conversion completed under each conversion sequence.

Threshold function.

- static void ADC_SetThresholdPair0 (ADC_Type *base, uint32_t lowValue, uint32_t highValue)
  Set the threshhold pair 0 with low and high value.
- static void ADC_SetThresholdPair1 (ADC_Type *base, uint32_t lowValue, uint32_t highValue)
  Set the threshhold pair 1 with low and high value.
- static void ADC_SetChannelWithThresholdPair0 (ADC_Type *base, uint32_t channelMask)
  Set given channels to apply the threshold pare 0.
- static void ADC_SetChannelWithThresholdPair1 (ADC_Type *base, uint32_t channelMask)
  Set given channels to apply the threshold pare 1.

Interrupts.

- static void ADC_EnableInterrupts (ADC_Type *base, uint32_t mask)
  Enable interrupts for conversion sequences.
- static void ADC_DisableInterrupts (ADC_Type *base, uint32_t mask)
  Disable interrupts for conversion sequence.
- static void ADC_EnableShresholdCompareInterrupt (ADC_Type *base, uint32_t channel, adc_threshold_interrupt_mode_t mode)
  Enable the interrupt of threshold compare event for each channel.
- static void ADC_EnableThresholdCompareInterrupt (ADC_Type *base, uint32_t channel, adc_threshold_interrupt_mode_t mode)
  Enable the interrupt of threshold compare event for each channel.

Status.

- static uint32_t ADC_GetStatusFlags (ADC_Type *base)
  Get status flags of ADC module.
- static void ADC_ClearStatusFlags (ADC_Type *base, uint32_t mask)
  Clear status flags of ADC module.

11.3 Data Structure Documentation

11.3.1 struct adc_config_t

Data Fields

- adc_clock_mode_t clockMode
  Select the clock mode for ADC converter.
- uint32_t clockDividerNumber
  This field is only available when using kADC_ClockSynchronousMode for "clockMode" field.
- bool enableLowPowerMode
  `If disable low-power mode, ADC remains activated even when no conversions are` *requested.*
- adc_vdda_range_t voltageRange
  Configure the ADC for the appropriate operating range of the analog supply voltage VDDA.

11.3.1.0.0.5 Field Documentation

11.3.1.0.0.5.1 adc_clock_mode_t adc_config_t::clockMode

11.3.1.0.0.5.2 uint32_t adc_config_t::clockDividerNumber

The divider would be plused by 1 based on the value in this field. The available range is in 8 bits.

11.3.1.0.0.5.3 bool adc_config_t::enableLowPowerMode

If enable low-power mode, The ADC is automatically powered-down when no conversions are taking place.

11.3.1.0.0.5.4 adc_vdda_range_t adc_config_t::voltageRange

Failure to set the area correctly causes the ADC to return incorrect conversion results.

11.3.2 struct adc_conv_seq_config_t

Data Fields

- uint32_t channelMask
  `Selects which one or more of the ADC channels will be sampled and conver`
  sequence is launched.
- uint32_t triggerMask
  `Selects which one or more of the available hardware trigger sources will`
  conversion sequence to be initiated.
- adc_trigger_polarity_t triggerPolarity
  Select the trigger to lauch conversion sequence.
- bool enableSyncBypass
  `To enable this feature allows the hardware trigger input to bypass synchr`
  flip-flop stages and therefore shorten the time between the trigger input signal and the start of a conversion.
- bool enableSingleStep
  `When enabling this feature, a trigger will launch a single conversion on`
  channel in the sequence instead of the default response of launching an entire sequence of conversions.
- adc_seq_interrupt_mode_t interruptMode
  Select the interrpt/DMA trigger mode.

11.3.2.0.0.6 Field Documentation

11.3.2.0.0.6.1 uint32_t adc_conv_seq_config_t::channelMask

The masked channels would be involved in current conversion sequence, beginning with the lowest-order. The available range is in 12-bit.

11.3.2.0.0.6.2 uint32_t adc_conv_seq_config_t::triggerMask

The available range is 6-bit.

11.3.2.0.0.6.3 adc_trigger_polarity_t adc_conv_seq_config_t::triggerPolarity

11.3.2.0.0.6.4 bool adc_conv_seq_config_t::enableSyncBypass

11.3.2.0.0.6.5 bool adc_conv_seq_config_t::enableSingleStep

11.3.2.0.0.6.6 adc_seq_interrupt_mode_t adc_conv_seq_config_t::interruptMode

11.3.3 struct adc_result_info_t

Data Fields

- uint32_t result
  Keep the conversion data value.
- adc_threshold_compare_status_t thresholdCompareStatus
  Keep the threshold compare status.
- adc_threshold_crossing_status_t thresholdCorssingStatus
  Keep the threshold crossing status.
- uint32_t channelNumber
  Keep the channel number for this conversion.
- bool overrunFlag
  Keep the status whether the conversion is overrun or not.

11.3.3.0.0.7 Field Documentation

11.3.3.0.0.7.1 uint32_t adc_result_info_t::result

11.3.3.0.0.7.2 adc_threshold_compare_status_t adc_result_info_t::thresholdCompareStatus

11.3.3.0.0.7.3 adc_threshold_crossing_status_t adc_result_info_t::thresholdCorssingStatus

11.3.3.0.0.7.4 uint32_t adc_result_info_t::channelNumber

11.3.3.0.0.7.5 bool adc_result_info_t::overrunFlag

11.4 Macro Definition Documentation

11.4.1 #define FSL_ADC_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))

11.5 Enumeration Type Documentation

11.5.1 enum _adc_status_flags

Enumerator

kADC_ThresholdCompareFlagOnChn0	Threshold comparison event on Channel 0.
kADC_ThresholdCompareFlagOnChn1	Threshold comparison event on Channel 1.
kADC_ThresholdCompareFlagOnChn2	Threshold comparison event on Channel 2.
kADC_ThresholdCompareFlagOnChn3	Threshold comparison event on Channel 3.
kADC_ThresholdCompareFlagOnChn4	Threshold comparison event on Channel 4.

kADC_ThresholdCompareFlagOnChn5 Threshold comparison event on Channel 5.
kADC_ThresholdCompareFlagOnChn6 Threshold comparison event on Channel 6.
kADC_ThresholdCompareFlagOnChn7 Threshold comparison event on Channel 7.
kADC_ThresholdCompareFlagOnChn8 Threshold comparison event on Channel 8.
kADC_ThresholdCompareFlagOnChn9 Threshold comparison event on Channel 9.
kADC_ThresholdCompareFlagOnChn10 Threshold comparison event on Channel 10.
kADC_ThresholdCompareFlagOnChn11 Threshold comparison event on Channel 11.
kADC_OverrunFlagForChn0 Mirror the OVERRUN status flag from the result register for ADC channel 0.
kADC_OverrunFlagForChn1 Mirror the OVERRUN status flag from the result register for ADC channel 1.
kADC_OverrunFlagForChn2 Mirror the OVERRUN status flag from the result register for ADC channel 2.
kADC_OverrunFlagForChn3 Mirror the OVERRUN status flag from the result register for ADC channel 3.
kADC_OverrunFlagForChn4 Mirror the OVERRUN status flag from the result register for ADC channel 4.
kADC_OverrunFlagForChn5 Mirror the OVERRUN status flag from the result register for ADC channel 5.
kADC_OverrunFlagForChn6 Mirror the OVERRUN status flag from the result register for ADC channel 6.
kADC_OverrunFlagForChn7 Mirror the OVERRUN status flag from the result register for ADC channel 7.
kADC_OverrunFlagForChn8 Mirror the OVERRUN status flag from the result register for ADC channel 8.
kADC_OverrunFlagForChn9 Mirror the OVERRUN status flag from the result register for ADC channel 9.
kADC_OverrunFlagForChn10 Mirror the OVERRUN status flag from the result register for ADC channel 10.
kADC_OverrunFlagForChn11 Mirror the OVERRUN status flag from the result register for ADC channel 11.
kADC_GlobalOverrunFlagForSeqA Mirror the glabal OVERRUN status flag for conversion sequence A.
kADC_GlobalOverrunFlagForSeqB Mirror the global OVERRUN status flag for conversion sequence B.
kADC_ConvSeqAInterruptFlag Sequence A interrupt/DMA trigger.
kADC_ConvSeqBInterruptFlag Sequence B interrupt/DMA trigger.
kADC_ThresholdCompareInterruptFlag Threshold comparision interrupt flag.
kADC_OverrunInterruptFlag Overrun interrupt flag.

11.5.2 enum _adc_interrupt_enable

Note

Not all the interrupt options are listed here

Enumerator

kADC_ConvSeqAInterruptEnable Enable interrupt upon completion of each individual conversion in sequence A, or entire sequence.
kADC_ConvSeqBInterruptEnable Enable interrupt upon completion of each individual conversion in sequence B, or entire sequence.
kADC_OverrunInterruptEnable Enable the detection of an overrun condition on any of the channel data registers will cause an overrun interrupt/DMA trigger.

11.5.3 enum adc_clock_mode_t

Enumerator

kADC_ClockSynchronousMode The ADC clock would be derived from the system clock based on "clockDividerNumber".
kADC_ClockAsynchronousMode The ADC clock would be based on the SYSCON block's divider.

11.5.4 enum adc_trigger_polarity_t

Enumerator

kADC_TriggerPolarityNegativeEdge A negative edge launches the conversion sequence on the trigger(s).
kADC_TriggerPolarityPositiveEdge A positive edge launches the conversion sequence on the trigger(s).

11.5.5 enum adc_priority_t

Enumerator

kADC_PriorityLow This sequence would be preempted when another sequence is started.
kADC_PriorityHigh This sequence would preempt other sequence even when it is started.

11.5.6 enum adc_seq_interrupt_mode_t

Enumerator

kADC_InterruptForEachConversion The sequence interrupt/DMA trigger will be set at the end of each individual ADC conversion inside this conversion sequence.

kADC_InterruptForEachSequence The sequence interrupt/DMA trigger will be set when the entire set of this sequence conversions completes.

11.5.7 enum adc_threshold_compare_status_t

Enumerator

kADC_ThresholdCompareInRange LOW threshold <= conversion value <= HIGH threshold.
kADC_ThresholdCompareBelowRange conversion value < LOW threshold.
kADC_ThresholdCompareAboveRange conversion value > HIGH threshold.

11.5.8 enum adc_threshold_crossing_status_t

Enumerator

kADC_ThresholdCrossingNoDetected No threshold Crossing detected.
kADC_ThresholdCrossingDownward Downward Threshold Crossing detected.
kADC_ThresholdCrossingUpward Upward Threshold Crossing Detected.

11.5.9 enum adc_threshold_interrupt_mode_t

Enumerator

kADC_ThresholdInterruptDisabled Threshold comparison interrupt is disabled.
kADC_ThresholdInterruptOnOutside Threshold comparison interrupt is enabled on outside threshold.
kADC_ThresholdInterruptOnCrossing Threshold comparison interrupt is enabled on crossing threshold.

11.5.10 enum adc_inforesult_t

Enumerator

kADC_Resolution12bitInfoResultShift Info result shift of Resolution12bit.
kADC_Resolution10bitInfoResultShift Info result shift of Resolution10bit.
kADC_Resolution8bitInfoResultShift Info result shift of Resolution8bit.
kADC_Resolution6bitInfoResultShift Info result shift of Resolution6bit.

11.5.11 enum adc_tempsensor_common_mode_t

Enumerator

kADC_HighNegativeOffsetAdded Temerature sensor common mode: high negative offset added.
kADC_IntermediateNegativeOffsetAdded Temerature sensor common mode: intermediate negative offset added.
kADC_NoOffsetAdded Temerature sensor common mode: no offset added.
kADC_LowPositiveOffsetAdded Temerature sensor common mode: low positive offset added.

11.5.12 enum adc_second_control_t

Enumerator

kADC_Impedance621Ohm Extand ADC sampling time according to source impedance 1: 0.621 kOhm.
kADC_Impedance55kOhm Extand ADC sampling time according to source impedance 20 (default): 55 kOhm.
kADC_Impedance87kOhm Extand ADC sampling time according to source impedance 31: 87 k-Ohm.
kADC_NormalFunctionalMode TEST mode: Normal functional mode.
kADC_MultiplexeTestMode TEST mode: Multiplexer test mode.
kADC_ADCInUnityGainMode TEST mode: ADC in unity gain mode.

11.6 Function Documentation

11.6.1 void ADC_Init (ADC_Type * *base*, const adc_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_config_t.

11.6.2 void ADC_Deinit (ADC_Type * *base*)

Parameters

base	ADC peripheral base address.

11.6.3 void ADC_GetDefaultConfig (adc_config_t * config)

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

```
*   config->clockMode = kADC_ClockSynchronousMode;
*   config->clockDividerNumber = 0U;
*   config->resolution = kADC_Resolution12bit;
*   config->enableBypassCalibration = false;
*   config->sampleTimeNumber = 0U;
*
```

Parameters

config	Pointer to configuration structure.

11.6.4 bool ADC_DoSelfCalibration (ADC_Type * base, uint32_t frequency)

To calibrate the ADC, set the ADC clock to 500 kHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

base	ADC peripheral base address.
frequency	The ststem clock frequency to ADC.

Return values

true	Calibration succeed.
false	Calibration failed.

11.6.5 static void ADC_EnableConvSeqA (ADC_Type * base, bool enable) [inline], [static]

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable the feature or not.

11.6.6 void ADC_SetConvSeqAConfig (ADC_Type * *base*, const adc_conv_seq_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_conv_seq_config_t.

11.6.7 static void ADC_DoSoftwareTriggerConvSeqA (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

11.6.8 static void ADC_EnableConvSeqABurstMode (ADC_Type * *base*, bool *enable*) [inline], [static]

Enable the burst mode would cause the conversion sequence to be cntinuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before cnversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable this feature.

11.6.9 static void ADC_SetConvSeqAHighPriority (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral bass address.

11.6.10 static void ADC_EnableConvSeqB (ADC_Type * *base*, bool *enable*) [inline], [static]

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable the feature or not.

11.6.11 void ADC_SetConvSeqBConfig (ADC_Type * *base*, const adc_conv_seq_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_conv_seq_config_t.

11.6.12 static void ADC_DoSoftwareTriggerConvSeqB (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

11.6.13 static void ADC_EnableConvSeqBBurstMode (ADC_Type * *base*, bool *enable*) [inline], [static]

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before cnversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable this feature.

11.6.14 static void ADC_SetConvSeqBHighPriority (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral bass address.

11.6.15 bool ADC_GetConvSeqAGlobalConversionResult (ADC_Type * *base*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

11.6.16 bool ADC_GetConvSeqBGlobalConversionResult (ADC_Type * *base*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

11.6.17 bool ADC_GetChannelConversionResult (ADC_Type * *base*, uint32_t *channel*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
channel	The indicated channel number.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

11.6.18 static void ADC_SetThresholdPair0 (ADC_Type * *base*, uint32_t *lowValue*, uint32_t *highValue*) [inline], [static]

Parameters

base	ADC peripheral base address.
lowValue	LOW threshold value.
highValue	HIGH threshold value.

11.6.19 static void ADC_SetThresholdPair1 (ADC_Type * *base*, uint32_t *lowValue*, uint32_t *highValue*) [inline], [static]

Parameters

base	ADC peripheral base address.
lowValue	LOW threshold value. The available value is with 12-bit.
highValue	HIGH threshold value. The available value is with 12-bit.

11.6.20 static void ADC_SetChannelWithThresholdPair0 (ADC_Type ∗ *base,* uint32_t *channelMask*) [inline], [static]

Parameters

base	ADC peripheral base address.
channelMask	Indicated channels' mask.

11.6.21 static void ADC_SetChannelWithThresholdPair1 (ADC_Type ∗ *base,* uint32_t *channelMask*) [inline], [static]

Parameters

base	ADC peripheral base address.
channelMask	Indicated channels' mask.

11.6.22 static void ADC_EnableInterrupts (ADC_Type ∗ *base,* uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of interrupt mask value for global block except each channal, see to _adc_-interrupt_enable.

11.6.23 static void ADC_DisableInterrupts (ADC_Type ∗ *base,* uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of interrupt mask value for global block except each channel, see to _adc_-interrupt_enable.

11.6.24 static void ADC_EnableShresholdCompareInterrupt (ADC_Type * *base*, uint32_t *channel*, adc_threshold_interrupt_mode_t *mode*) [inline], [static]

11.6.25 static void ADC_EnableThresholdCompareInterrupt (ADC_Type * *base*, uint32_t *channel*, adc_threshold_interrupt_mode_t *mode*) [inline], [static]

Parameters

base	ADC peripheral base address.
channel	Channel number.
mode	Interrupt mode for threshold compare event, see to adc_threshold_interrupt_mode_t.

11.6.26 static uint32_t ADC_GetStatusFlags (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

Returns

Mask of status flags of module, see to _adc_status_flags.

11.6.27 static void ADC_ClearStatusFlags (ADC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of status flags of module, see to _adc_status_flags.

Chapter 12
DAC: 10-bit Digital To Analog Converter Driver

12.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 10-bit digital to analog converter (DAC) module of MCUXpresso SDK devices.

12.2 Typical use case

12.2.1 Polling Configuration

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

12.2.2 Interrupt Configuration

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

Files

- file fsl_dac.h

Data Structures

- struct dac_config_t
  The configuration of DAC. More...

Enumerations

- enum dac_settling_time_t {
  kDAC_SettlingTimeIs1us = 0U,
  kDAC_SettlingTimeIs25us = 1U }
  The DAC settling time.

Functions

- void DAC_Init (DAC_Type ∗base, const dac_config_t ∗config)
  Initialize the DAC module.
- void DAC_Deinit (DAC_Type ∗base)
  De-Initialize the DAC module.
- void DAC_GetDefaultConfig (dac_config_t ∗config)
  Initializes the DAC user configuration structure.
- void DAC_EnableDoubleBuffering (DAC_Type ∗base, bool enable)

Enable/Diable double-buffering feature.
- void DAC_SetBufferValue (DAC_Type ∗base, uint32_t value)
  Write DAC output value into CR register or pre-buffer.
- void DAC_SetCounterValue (DAC_Type ∗base, uint32_t value)
  Write DAC counter value into CNTVAL register.
- static void DAC_EnableDMA (DAC_Type ∗base, bool enable)
  Enable/Disable the DMA access.
- static void DAC_EnableCounter (DAC_Type ∗base, bool enable)
  Enable/Disable the counter operation.
- static bool DAC_GetDMAInterruptRequestFlag (DAC_Type ∗base)
  Get the status flag of DMA or interrupt request.

Driver version

- #define LPC_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  DAC driver version 2.0.1.

12.3 Data Structure Documentation

12.3.1 struct dac_config_t

Data Fields

- dac_settling_time_t settlingTime
  The settling times are valid for a capacitance load on the DAC_OUT pin not exceeding 100 pF.

12.3.1.0.0.8 Field Documentation

12.3.1.0.0.8.1 dac_settling_time_t dac_config_t::settlingTime

A load impedance value greater than that value will cause settling time longer than the specified time. One or more graphs of load impedance vs. settling time will be included in the final data sheet.

12.4 Macro Definition Documentation

12.4.1 #define LPC_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

12.5 Enumeration Type Documentation

12.5.1 enum dac_settling_time_t

Enumerator

kDAC_SettlingTimeIs1us The settling time of the DAC is 1us max, and the maximum current is 700 mA. This allows a maximum update rate of 1 MHz.

kDAC_SettlingTimeIs25us The settling time of the DAC is 2.5us and the maximum current is 350uA. This allows a maximum update rate of 400 kHz.

12.6 Function Documentation

12.6.1 void DAC_Init (DAC_Type * *base*, const dac_config_t * *config*)

Parameters

base	DAC peripheral base address.
config	The pointer to configuration structure. Please refer to "dac_config_t" structure.

12.6.2 void DAC_Deinit (DAC_Type * *base*)

Parameters

base	DAC peripheral base address.

12.6.3 void DAC_GetDefaultConfig (dac_config_t * *config*)

This function initializes the user configuration structure to a default value. The default values are as follows.

```
*   config->settlingTime = kDAC_SettlingTimeIs1us;
*
```

Parameters

config	Pointer to the configuration structure. See "dac_config_t".

12.6.4 void DAC_EnableDoubleBuffering (DAC_Type * *base,* bool *enable*)

Notice: Disabling the double-buffering feature will disable counter opreation. If double-buffering feature is disabled, any writes to the CR address will go directly to the CR register. If double-buffering feature is enabled, any write to the CR register will only load the pre-buffer, which shares its register address with the CR register. The CR itself will be loaded from the pre-buffer whenever the counter reaches zero and the DMA request is set.

Parameters

base	DAC peripheral base address.

enable	Enable or disable the feature.

12.6.5 void DAC_SetBufferValue (DAC_Type * *base*, uint32_t *value*)

The DAC output voltage is VALUE*((VREFP)/1024).

Parameters

base	DAC peripheral base address.
value	Setting the value for items in the buffer. 10-bits are available.

12.6.6 void DAC_SetCounterValue (DAC_Type * *base*, uint32_t *value*)

```
When the counter is enabled bit, the 16-bit counter will begin counting down, at the rate s
from the value programmed into the DACCNTVAL register. The counter is decremented Each time
```

reaches zero, the counter will be reloaded by the value of DACCNTVAL and the DMA request bit INT_-DMA_REQ will be set in hardware.

Parameters

base	DAC peripheral basic address.
value	Setting the value for items in the counter. 16-bits are available.

12.6.7 static void DAC_EnableDMA (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	DAC peripheral base address.
enable	Enable or disable the feature.

12.6.8 static void DAC_EnableCounter (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	DAC peripheral base address.
enable	Enable or disable the feature.

12.6.9 static bool DAC_GetDMAInterruptRequestFlag (DAC_Type * *base*) [inline], [static]

Parameters

base	DAC peripheral base address.

Returns

If return 'true', it means DMA request or interrupt occurs. If return 'false', it means DMA request or interrupt doesn't occur.

Chapter 13
LPC_ACOMP: Analog comparator Driver

13.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Analog comparator (LPC_ACOMP) module of MCUXpresso SDK devices.

13.2 Typical use case

13.2.1 Polling Configuration

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

13.2.2 Interrupt Configuration

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

Files

- file fsl_acomp.h

Data Structures

- struct acomp_config_t
  The structure for ACOMP basic configuration. More...
- struct acomp_ladder_config_t
  The structure for ACOMP voltage ladder. More...

Enumerations

- enum acomp_ladder_reference_voltage_t {
  kACOMP_LadderRefVoltagePinVDD = 0U,
  kACOMP_LadderRefVoltagePinVDDCMP = 1U }
- enum acomp_interrupt_enable_t {
  kACOMP_InterruptsFallingEdgeEnable = 0U,
  kACOMP_InterruptsRisingEdgeEnable = 1U,
  kACOMP_InterruptsBothEdgesEnable = 2U,
  kACOMP_InterruptsDisable = 3U }
  The ACOMP interrupts enable.

- enum acomp_hysteresis_selection_t {
kACOMP_HysteresisNoneSelection = 0U,
kACOMP_Hysteresis5MVSelection = 1U,
kACOMP_Hysteresis10MVSelection = 2U,
kACOMP_Hysteresis15MVSelection = 3U }
The ACOMP hysteresis selection.

Variables

- bool acomp_config_t::enableSyncToBusClk
If true, Comparator output is synchronized to the bus clock for output to other modules.
- acomp_hysteresis_selection_t acomp_config_t::hysteresisSelection
Controls the hysteresis of the comparator.
- uint8_t acomp_ladder_config_t::ladderValue
Voltage ladder value.
- acomp_ladder_reference_voltage_t acomp_ladder_config_t::referenceVoltage
Selects the reference voltage(Vref) for the voltage ladder.

Driver version

- #define FSL_ACOMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
ACOMP driver version 2.0.2.

Initialization

- void ACOMP_Init (ACOMP_Type ∗base, const acomp_config_t ∗config)
Initialize the ACOMP module.
- void ACOMP_Deinit (ACOMP_Type ∗base)
De-initialize the ACOMP module.
- void ACOMP_GetDefaultConfig (acomp_config_t ∗config)
Gets an available pre-defined settings for the ACOMP's configuration.
- void ACOMP_EnableInterrupts (ACOMP_Type ∗base, acomp_interrupt_enable_t enable)
Enable ACOMP interrupts.
- static bool ACOMP_GetInterruptsStatusFlags (ACOMP_Type ∗base)
Get interrupts status flags.
- static void ACOMP_ClearInterruptsStatusFlags (ACOMP_Type ∗base)
Clear the ACOMP interrupts status flags.
- static bool ACOMP_GetOutputStatusFlags (ACOMP_Type ∗base)
Get ACOMP output status flags.
- static void ACOMP_SetInputChannel (ACOMP_Type ∗base, uint32_t postiveInputChannel, uint32-_t negativeInputChannel)
Set the ACOMP postive and negative input channel.
- void ACOMP_SetLadderConfig (ACOMP_Type ∗base, const acomp_ladder_config_t ∗config)
Set the voltage ladder configuration.

13.3 Data Structure Documentation

13.3.1 struct acomp_config_t

Data Fields

- bool enableSyncToBusClk
  If true, Comparator output is synchronized to the bus clock for output to other modules.
- acomp_hysteresis_selection_t hysteresisSelection
  Controls the hysteresis of the comparator.

13.3.2 struct acomp_ladder_config_t

Data Fields

- uint8_t ladderValue
  Voltage ladder value.
- acomp_ladder_reference_voltage_t referenceVoltage
  Selects the reference voltage(Vref) for the voltage ladder.

13.4 Macro Definition Documentation

13.4.1 #define FSL_ACOMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

13.5 Enumeration Type Documentation

13.5.1 enum acomp_ladder_reference_voltage_t

Enumerator

kACOMP_LadderRefVoltagePinVDD Supply from pin VDD.
kACOMP_LadderRefVoltagePinVDDCMP Supply from pin VDDCMP.

13.5.2 enum acomp_interrupt_enable_t

Enumerator

kACOMP_InterruptsFallingEdgeEnable Enable the falling edge interrupts.
kACOMP_InterruptsRisingEdgeEnable Enable the rising edge interrupts.
kACOMP_InterruptsBothEdgesEnable Enable the both edges interrupts.
kACOMP_InterruptsDisable Disable the interrupts.

13.5.3 enum acomp_hysteresis_selection_t

Enumerator

kACOMP_HysteresisNoneSelection None (the output will switch as the voltages cross).
kACOMP_Hysteresis5MVSelection 5Mv.
kACOMP_Hysteresis10MVSelection 10Mv.
kACOMP_Hysteresis15MVSelection 15Mv.

13.6 Function Documentation

13.6.1 void ACOMP_Init (ACOMP_Type * *base*, const acomp_config_t * *config*)

Parameters

base	ACOMP peripheral base address.
config	Pointer to "acomp_config_t" structure.

13.6.2 void ACOMP_Deinit (ACOMP_Type * *base*)

Parameters

base	ACOMP peripheral base address.

13.6.3 void ACOMP_GetDefaultConfig (acomp_config_t * *config*)

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

```
*   config->enableSyncToBusClk = false;
*   config->hysteresisSelection = kACOMP_hysteresisNoneSelection;
*
```

In default configuration, the ACOMP's output would be used directly and switch as the voltages cross.

Parameters

base	ACOMP peripheral base address.
config	Pointer to the configuration structure.

13.6.4 void ACOMP_EnableInterrupts (ACOMP_Type * *base*, acomp_interrupt_enable_t *enable*)

Parameters

base	ACOMP peripheral base address.
enable	Enable/Disable interrupt feature.

13.6.5 static bool ACOMP_GetInterruptsStatusFlags (ACOMP_Type * *base*) [inline], [static]

Parameters

base	ACOMP peripheral base address.

Returns

Reflect the state ACOMP edge-detect status, true or false.

13.6.6 static void ACOMP_ClearInterruptsStatusFlags (ACOMP_Type * *base*) [inline], [static]

Parameters

base	ACOMP peripheral base address.

13.6.7 static bool ACOMP_GetOutputStatusFlags (ACOMP_Type * *base*) [inline], [static]

Parameters

base	ACOMP peripheral base address.

Returns

Reflect the state of the comparator output, true or false.

13.6.8 static void ACOMP_SetInputChannel (ACOMP_Type ∗ *base,* uint32_t *postiveInputChannel,* uint32_t *negativeInputChannel*) [inline], [static]

Parameters

base	ACOMP peripheral base address.
postiveInput-Channel	The index of postive input channel.
negativeInput-Channel	The index of negative input channel.

13.6.9 void ACOMP_SetLadderConfig (ACOMP_Type ∗ *base,* const acomp_ladder_config_t ∗ *config*)

Parameters

base	ACOMP peripheral base address.
config	The structure for voltage ladder. If the config is NULL, voltage ladder would be diasbled, otherwise the voltage ladder would be configured and enabled.

13.7 Variable Documentation

13.7.1 bool acomp_config_t::enableSyncToBusClk

If false, Comparator output is used directly.

13.7.2 acomp_hysteresis_selection_t acomp_config_t::hysteresisSelection

13.7.3 uint8_t acomp_ladder_config_t::ladderValue

00000 = Vss, 00001 = 1∗Vref/31, ..., 11111 = Vref.

13.7.4 acomp_ladder_reference_voltage_t acomp_ladder_config_t::referenceVoltage

Chapter 14
DMA: Direct Memory Access Controller Driver

14.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Direct Memory Access (DMA) of MCUXpresso SDK devices.

14.2 Typical use case

14.2.1 DMA Operation

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

Files

- file fsl_dma.h

Data Structures

- struct dma_descriptor_t
  DMA descriptor structure. More...
- struct dma_xfercfg_t
  DMA transfer configuration. More...
- struct dma_channel_trigger_t
  DMA channel trigger. More...
- struct dma_channel_config_t
  DMA channel trigger. More...
- struct dma_transfer_config_t
  DMA transfer configuration. More...
- struct dma_handle_t
  DMA transfer handle structure. More...

Macros

- #define DMA_MAX_TRANSFER_COUNT 0x400
  DMA max transfer size.
- #define FSL_FEATURE_DMA_NUMBER_OF_CHANNELSn(x) FSL_FEATURE_DMA_NUMBER_OF_CHANNELS
  DMA channel numbers.
- #define FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE (16U)
  DMA head link descriptor table align size.
- #define DMA_ALLOCATE_HEAD_DESCRIPTORS(name, number) SDK_ALIGN(dma_descriptor_t name[number], FSL_FEATURE_DMA_DESCRIPTOR_ALIGN_SIZE)
  DMA head descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

- #define DMA_ALLOCATE_LINK_DESCRIPTORS(name, number) SDK_ALIGN(dma_descriptor_t name[number], FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE)
  DMA link descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.
- #define DMA_COMMON_REG_GET(base, channel, reg) (((volatile uint32_t ∗)(&((base)->COMMON[0].reg)))[DMA_CHANNEL_GROUP(channel)])
  DMA linked descriptor address algin size.
- #define DMA_DESCRIPTOR_END_ADDRESS(start, inc, bytes, width) ((void ∗)((uint32_t)(start) + inc ∗ bytes - inc ∗ width))
  DMA descriptor end address calculate.
- #define DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width, srcInc, dstInc, bytes)
  DMA channel transfer configurations macro.

Typedefs

- typedef void(∗ dma_callback)(struct _dma_handle ∗handle, void ∗userData, bool transferDone, uint32_t intmode)
  Define Callback function for DMA.

Enumerations

- enum _dma_transfer_status { kStatus_DMA_Busy = MAKE_STATUS(kStatusGroup_DMA, 0) }
  DMA transfer status.
- enum _dma_addr_interleave_size {
  kDMA_AddressInterleave0xWidth = 0U,
  kDMA_AddressInterleave1xWidth = 1U,
  kDMA_AddressInterleave2xWidth = 2U,
  kDMA_AddressInterleave4xWidth = 4U }
  dma address interleave size
- enum _dma_transfer_width {
  kDMA_Transfer8BitWidth = 1U,
  kDMA_Transfer16BitWidth = 2U,
  kDMA_Transfer32BitWidth = 4U }
  dma transfer width
- enum dma_priority_t {
  kDMA_ChannelPriority0 = 0,
  kDMA_ChannelPriority1,
  kDMA_ChannelPriority2,
  kDMA_ChannelPriority3,
  kDMA_ChannelPriority4,
  kDMA_ChannelPriority5,
  kDMA_ChannelPriority6,
  kDMA_ChannelPriority7 }
  DMA channel priority.
- enum dma_irq_t {
  kDMA_IntA,
  kDMA_IntB,
  kDMA_IntError }

DMA interrupt flags.
- enum dma_trigger_type_t {
kDMA_NoTrigger = 0,
kDMA_LowLevelTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1) | DMA_CHANNEL_CFG-_TRIGTYPE(1),
kDMA_HighLevelTrigger,
kDMA_FallingEdgeTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1),
kDMA_RisingEdgeTrigger }
DMA trigger type.
- enum _dma_burst_size {
kDMA_BurstSize1 = 0U,
kDMA_BurstSize2 = 1U,
kDMA_BurstSize4 = 2U,
kDMA_BurstSize8 = 3U,
kDMA_BurstSize16 = 4U,
kDMA_BurstSize32 = 5U,
kDMA_BurstSize64 = 6U,
kDMA_BurstSize128 = 7U,
kDMA_BurstSize256 = 8U,
kDMA_BurstSize512 = 9U,
kDMA_BurstSize1024 = 10U }
DMA burst size.
- enum dma_trigger_burst_t {
kDMA_SingleTransfer = 0,
kDMA_LevelBurstTransfer = DMA_CHANNEL_CFG_TRIGBURST(1),
kDMA_EdgeBurstTransfer1 = DMA_CHANNEL_CFG_TRIGBURST(1),
kDMA_EdgeBurstTransfer2,
kDMA_EdgeBurstTransfer4,
kDMA_EdgeBurstTransfer8,
kDMA_EdgeBurstTransfer16,
kDMA_EdgeBurstTransfer32,
kDMA_EdgeBurstTransfer64,
kDMA_EdgeBurstTransfer128,
kDMA_EdgeBurstTransfer256,
kDMA_EdgeBurstTransfer512,
kDMA_EdgeBurstTransfer1024 }
DMA trigger burst.
- enum dma_burst_wrap_t {
kDMA_NoWrap = 0,
kDMA_SrcWrap = DMA_CHANNEL_CFG_SRCBURSTWRAP(1),
kDMA_DstWrap = DMA_CHANNEL_CFG_DSTBURSTWRAP(1),
kDMA_SrcAndDstWrap }
DMA burst wrapping.
- enum dma_transfer_type_t {

kDMA_MemoryToMemory = 0x0U,
kDMA_PeripheralToMemory,
kDMA_MemoryToPeripheral,
kDMA_StaticToStatic }
DMA transfer type.

Driver version

- #define FSL_DMA_DRIVER_VERSION (MAKE_VERSION(2, 3, 0))
  DMA driver version.

DMA initialization and De-initialization

- void DMA_Init (DMA_Type ∗base)
  Initializes DMA peripheral.
- void DMA_Deinit (DMA_Type ∗base)
  Deinitializes DMA peripheral.
- void DMA_InstallDescriptorMemory (DMA_Type ∗base, void ∗addr)
  Install DMA descriptor memory.

DMA Channel Operation

- static bool DMA_ChannelIsActive (DMA_Type ∗base, uint32_t channel)
  Return whether DMA channel is processing transfer.
- static void DMA_EnableChannelInterrupts (DMA_Type ∗base, uint32_t channel)
  Enables the interrupt source for the DMA transfer.
- static void DMA_DisableChannelInterrupts (DMA_Type ∗base, uint32_t channel)
  Disables the interrupt source for the DMA transfer.
- static void DMA_EnableChannel (DMA_Type ∗base, uint32_t channel)
  Enable DMA channel.
- static void DMA_DisableChannel (DMA_Type ∗base, uint32_t channel)
  Disable DMA channel.
- static void DMA_EnableChannelPeriphRq (DMA_Type ∗base, uint32_t channel)
  Set PERIPHREQEN of channel configuration register.
- static void DMA_DisableChannelPeriphRq (DMA_Type ∗base, uint32_t channel)
  Get PERIPHREQEN value of channel configuration register.
- void DMA_ConfigureChannelTrigger (DMA_Type ∗base, uint32_t channel, dma_channel_trigger_t ∗trigger)
  Set trigger settings of DMA channel.
- void DMA_SetChannelConfig (DMA_Type ∗base, uint32_t channel, dma_channel_trigger_t ∗trigger, bool isPeriph)
  set channel config.
- uint32_t DMA_GetRemainingBytes (DMA_Type ∗base, uint32_t channel)
  Gets the remaining bytes of the current DMA descriptor transfer.
- static void DMA_SetChannelPriority (DMA_Type ∗base, uint32_t channel, dma_priority_t priority)
  Set priority of channel configuration register.
- static dma_priority_t DMA_GetChannelPriority (DMA_Type ∗base, uint32_t channel)
  Get priority of channel configuration register.
- static void DMA_SetChannelConfigValid (DMA_Type ∗base, uint32_t channel)
  Set channel configuration valid.

- static void DMA_DoChannelSoftwareTrigger (DMA_Type ∗base, uint32_t channel)
  Do software trigger for the channel.
- static void DMA_LoadChannelTransferConfig (DMA_Type ∗base, uint32_t channel, uint32_t xfer)
  Load channel transfer configurations.
- void DMA_CreateDescriptor (dma_descriptor_t ∗desc, dma_xfercfg_t ∗xfercfg, void ∗srcAddr, void ∗dstAddr, void ∗nextDesc)
  Create application specific DMA descriptor to be used in a chain in transfer.
- void DMA_SetupDescriptor (dma_descriptor_t ∗desc, uint32_t xfercfg, void ∗srcStartAddr, void ∗dstStartAddr, void ∗nextDesc)
  setup dma descriptor
- void DMA_SetupChannelDescriptor (dma_descriptor_t ∗desc, uint32_t xfercfg, void ∗srcStartAddr, void ∗dstStartAddr, void ∗nextDesc, dma_burst_wrap_t wrapType, uint32_t burstSize)
  setup dma channel descriptor

DMA Transactional Operation

- void DMA_AbortTransfer (dma_handle_t ∗handle)
  Abort running transfer by handle.
- void DMA_CreateHandle (dma_handle_t ∗handle, DMA_Type ∗base, uint32_t channel)
  Creates the DMA handle.
- void DMA_SetCallback (dma_handle_t ∗handle, dma_callback callback, void ∗userData)
  Installs a callback function for the DMA transfer.
- void DMA_PrepareTransfer (dma_transfer_config_t ∗config, void ∗srcAddr, void ∗dstAddr, uint32_t byteWidth, uint32_t transferBytes, dma_transfer_type_t type, void ∗nextDesc)
  Prepares the DMA transfer structure.
- void DMA_PrepareChannelTransfer (dma_channel_config_t ∗config, void ∗srcStartAddr, void ∗dstStartAddr, uint32_t xferCfg, dma_transfer_type_t type, dma_channel_trigger_t ∗trigger, void ∗nextDesc)
  Prepare channel transfer configurations.
- status_t DMA_SubmitTransfer (dma_handle_t ∗handle, dma_transfer_config_t ∗config)
  Submits the DMA transfer request.
- void DMA_SubmitChannelTransferParameter (dma_handle_t ∗handle, uint32_t xfercfg, void ∗srcStartAddr, void ∗dstStartAddr, void ∗nextDesc)
  Submit channel transfer paramter directly.
- void DMA_SubmitChannelDescriptor (dma_handle_t ∗handle, dma_descriptor_t ∗descriptor)
  Submit channel descriptor.
- status_t DMA_SubmitChannelTransfer (dma_handle_t ∗handle, dma_channel_config_t ∗config)
  Submits the DMA channel transfer request.
- void DMA_StartTransfer (dma_handle_t ∗handle)
  DMA start transfer.
- void DMA_IRQHandle (DMA_Type ∗base)
  DMA IRQ handler for descriptor transfer complete.

14.3 Data Structure Documentation

14.3.1 struct dma_descriptor_t

Data Fields

- volatile uint32_t xfercfg
  Transfer configuration.
- void * srcEndAddr
  Last source address of DMA transfer.
- void * dstEndAddr
  Last destination address of DMA transfer.
- void * linkToNextDesc
  Address of next DMA descriptor in chain.

14.3.2 struct dma_xfercfg_t

Data Fields

- bool valid
  Descriptor is ready to transfer.
- bool reload
  Reload channel configuration register after current descriptor is exhausted.
- bool swtrig
  Perform software trigger.
- bool clrtrig
  Clear trigger.
- bool intA
  Raises IRQ when transfer is done and set IRQA status register flag.
- bool intB
  Raises IRQ when transfer is done and set IRQB status register flag.
- uint8_t byteWidth
  Byte width of data to transfer.
- uint8_t srcInc
  Increment source address by 'srcInc' x 'byteWidth'.
- uint8_t dstInc
  Increment destination address by 'dstInc' x 'byteWidth'.
- uint16_t transferCount
  Number of transfers.

14.3.2.0.0.9 Field Documentation

14.3.2.0.0.9.1 bool dma_xfercfg_t::swtrig

Transfer if fired when 'valid' is set

14.3.3 struct dma_channel_trigger_t

Data Fields

- dma_trigger_type_t type
  Select hardware trigger as edge triggered or level triggered.
- dma_trigger_burst_t burst
  Select whether hardware triggers cause a single or burst transfer.
- dma_burst_wrap_t wrap
  Select wrap type, source wrap or dest wrap, or both.

14.3.3.0.0.10 Field Documentation

14.3.3.0.0.10.1 dma_trigger_type_t dma_channel_trigger_t::type

14.3.3.0.0.10.2 dma_trigger_burst_t dma_channel_trigger_t::burst

14.3.3.0.0.10.3 dma_burst_wrap_t dma_channel_trigger_t::wrap

14.3.4 struct dma_channel_config_t

Data Fields

- void ∗ srcStartAddr
  Source data address.
- void ∗ dstStartAddr
  Destination data address.
- void ∗ nextDesc
  Chain custom descriptor.
- uint32_t xferCfg
  channel transfer configurations
- dma_channel_trigger_t ∗ trigger
  DMA trigger type.
- bool isPeriph
  select the request type

14.3.5 struct dma_transfer_config_t

Data Fields

- uint8_t ∗ srcAddr
  Source data address.
- uint8_t ∗ dstAddr
  Destination data address.
- uint8_t ∗ nextDesc
  Chain custom descriptor.
- dma_xfercfg_t xfercfg

Transfer options.
- bool isPeriph
  DMA transfer is driven by peripheral.

14.3.6 struct dma_handle_t

Data Fields

- dma_callback callback
  Callback function.
- void * userData
  Callback function parameter.
- DMA_Type * base
  DMA peripheral base address.
- uint8_t channel
  DMA channel number.

14.3.6.0.0.11 Field Documentation

14.3.6.0.0.11.1 dma_callback dma_handle_t::callback

Invoked when transfer of descriptor with interrupt flag finishes

14.4 Macro Definition Documentation

14.4.1 #define FSL_DMA_DRIVER_VERSION (MAKE_VERSION(2, 3, 0))

Version 2.3.0.

14.4.2 #define DMA_ALLOCATE_HEAD_DESCRIPTORS(*name*, *number*) SDK_ALIGN(dma_descriptor_t name[number], FSL_FEATURE_DMA_DESCRIPTOR_ALIGN_SIZE)

Parameters

name,allocate	decriptor name.
number,number	of descriptor to be allocated.

14.4.3 #define DMA_ALLOCATE_LINK_DESCRIPTORS(*name*, *number*) SDK_ALIGN(dma_descriptor_t name[number], FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE)

Parameters

name,allocate	decriptor name.
number,number	of descriptor to be allocated.

14.4.4 #define DMA_DESCRIPTOR_END_ADDRESS(*start, inc, bytes, width*) ((void ∗)((uint32_t)(start) + inc ∗ bytes - inc ∗ width))

Parameters

start,start	address
inc,address	interleave size
bytes,transfer	bytes
width,transfer	width

14.4.5 #define DMA_CHANNEL_XFER(*reload, clrTrig, intA, intB, width, srcInc, dstInc, bytes*)

Value:

```
DMA_CHANNEL_XFERCFG_CFGVALID_MASK | DMA_CHANNEL_XFERCFG_RELOAD(reload) | DMA_CHANNEL_XFERCFG_CLRTRIG(
      clrTrig) | \
        DMA_CHANNEL_XFERCFG_SETINTA(intA) | DMA_CHANNEL_XFERCFG_SETINTB(intB) |
               \
        DMA_CHANNEL_XFERCFG_WIDTH(width == 4 ? 2 : (width - 1)) |
               \
        DMA_CHANNEL_XFERCFG_SRCINC(srcInc == 4 ? (srcInc - 1) : srcInc) |
               \
        DMA_CHANNEL_XFERCFG_DSTINC(dstInc == 4 ? (dstInc - 1) : dstInc) |
               \
        DMA_CHANNEL_XFERCFG_XFERCOUNT(bytes / width - 1)
```

Parameters

reload,true	is reload link descriptor after current exhaust, false is not
clrTrig,true	is clear trigger status, wait software trigger, false is not

intA,enable	interruptA
intB,enable	interruptB
width,transfer	width
srcInc,source	address interleave size
dst-Inc,destination	address interleave size
bytes,transfer	bytes

14.5 Typedef Documentation

14.5.1 typedef void(∗ dma_callback)(struct _dma_handle ∗handle, void ∗userData, bool transferDone, uint32_t intmode)

14.6 Enumeration Type Documentation

14.6.1 enum _dma_transfer_status

Enumerator

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

14.6.2 enum _dma_addr_interleave_size

Enumerator

kDMA_AddressInterleave0xWidth dma source/destination address no interleave
kDMA_AddressInterleave1xWidth dma source/destination address interleave 1xwidth
kDMA_AddressInterleave2xWidth dma source/destination address interleave 2xwidth
kDMA_AddressInterleave4xWidth dma source/destination address interleave 3xwidth

14.6.3 enum _dma_transfer_width

Enumerator

kDMA_Transfer8BitWidth dma channel transfer bit width is 8 bit
kDMA_Transfer16BitWidth dma channel transfer bit width is 16 bit
kDMA_Transfer32BitWidth dma channel transfer bit width is 32 bit

14.6.4 enum dma_priority_t

Enumerator

kDMA_ChannelPriority0 Highest channel priority - priority 0.
kDMA_ChannelPriority1 Channel priority 1.
kDMA_ChannelPriority2 Channel priority 2.
kDMA_ChannelPriority3 Channel priority 3.
kDMA_ChannelPriority4 Channel priority 4.
kDMA_ChannelPriority5 Channel priority 5.
kDMA_ChannelPriority6 Channel priority 6.
kDMA_ChannelPriority7 Lowest channel priority - priority 7.

14.6.5 enum dma_irq_t

Enumerator

kDMA_IntA DMA interrupt flag A.
kDMA_IntB DMA interrupt flag B.
kDMA_IntError DMA interrupt flag error.

14.6.6 enum dma_trigger_type_t

Enumerator

kDMA_NoTrigger Trigger is disabled.
kDMA_LowLevelTrigger Low level active trigger.
kDMA_HighLevelTrigger High level active trigger.
kDMA_FallingEdgeTrigger Falling edge active trigger.
kDMA_RisingEdgeTrigger Rising edge active trigger.

14.6.7 enum _dma_burst_size

Enumerator

kDMA_BurstSize1 burst size 1 transfer
kDMA_BurstSize2 burst size 2 transfer
kDMA_BurstSize4 burst size 4 transfer
kDMA_BurstSize8 burst size 8 transfer
kDMA_BurstSize16 burst size 16 transfer
kDMA_BurstSize32 burst size 32 transfer
kDMA_BurstSize64 burst size 64 transfer

kDMA_BurstSize128 burst size 128 transfer
kDMA_BurstSize256 burst size 256 transfer
kDMA_BurstSize512 burst size 512 transfer
kDMA_BurstSize1024 burst size 1024 transfer

14.6.8 enum dma_trigger_burst_t

Enumerator

kDMA_SingleTransfer Single transfer.
kDMA_LevelBurstTransfer Burst transfer driven by level trigger.
kDMA_EdgeBurstTransfer1 Perform 1 transfer by edge trigger.
kDMA_EdgeBurstTransfer2 Perform 2 transfers by edge trigger.
kDMA_EdgeBurstTransfer4 Perform 4 transfers by edge trigger.
kDMA_EdgeBurstTransfer8 Perform 8 transfers by edge trigger.
kDMA_EdgeBurstTransfer16 Perform 16 transfers by edge trigger.
kDMA_EdgeBurstTransfer32 Perform 32 transfers by edge trigger.
kDMA_EdgeBurstTransfer64 Perform 64 transfers by edge trigger.
kDMA_EdgeBurstTransfer128 Perform 128 transfers by edge trigger.
kDMA_EdgeBurstTransfer256 Perform 256 transfers by edge trigger.
kDMA_EdgeBurstTransfer512 Perform 512 transfers by edge trigger.
kDMA_EdgeBurstTransfer1024 Perform 1024 transfers by edge trigger.

14.6.9 enum dma_burst_wrap_t

Enumerator

kDMA_NoWrap Wrapping is disabled.
kDMA_SrcWrap Wrapping is enabled for source.
kDMA_DstWrap Wrapping is enabled for destination.
kDMA_SrcAndDstWrap Wrapping is enabled for source and destination.

14.6.10 enum dma_transfer_type_t

Enumerator

kDMA_MemoryToMemory Transfer from memory to memory (increment source and destination)
kDMA_PeripheralToMemory Transfer from peripheral to memory (increment only destination)
kDMA_MemoryToPeripheral Transfer from memory to peripheral (increment only source)
kDMA_StaticToStatic Peripheral to static memory (do not increment source or destination)

14.7 Function Documentation

14.7.1 void DMA_Init (DMA_Type ∗ *base*)

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

base	DMA peripheral base address.

14.7.2 void DMA_Deinit (DMA_Type ∗ *base*)

This function gates the DMA clock.

Parameters

base	DMA peripheral base address.

14.7.3 void DMA_InstallDescriptorMemory (DMA_Type ∗ *base,* void ∗ *addr*)

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

Parameters

base	DMA base address.
addr	DMA descriptor address

14.7.4 static bool DMA_ChannelIsActive (DMA_Type ∗ *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

True for active state, false otherwise.

14.7.5 static void DMA_EnableChannelInterrupts (DMA_Type ∗ *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

14.7.6 static void DMA_DisableChannelInterrupts (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

14.7.7 static void DMA_EnableChannel (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

True for enabled PeriphRq, false for disabled.

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

Parameters

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

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

This function provide a interface to configure channel configuration reisters.

Parameters

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

14.7.13 uint32_t DMA_GetRemainingBytes (DMA_Type * *base*, uint32_t *channel*)

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

The number of bytes which have not been transferred yet.

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

Parameters

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

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

Parameters

base	DMA peripheral base address.

channel	DMA channel number.

Returns

Channel priority value.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

14.7.18 static void DMA_LoadChannelTransferConfig (DMA_Type * *base*, uint32_t *channel*, uint32_t *xfer*) [inline], [static]

Parameters

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

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

Parameters

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

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

Note: This function do not support configure wrap descriptor.

Parameters

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

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

Note: This function support configure wrap descriptor.

Parameters

desc	DMA descriptor address.
xfercfg	Transfer configuration for DMA descriptor.
srcStartAddr	Start address of source address.
dstStartAddr	Start address of destination address.

nextDesc	Address of next descriptor in chain.
wrapType	burst wrap type.
burstSize	burst size, reference _dma_burst_size.

14.7.22 void DMA_AbortTransfer (dma_handle_t * *handle*)

This function aborts DMA transfer specified by handle.

Parameters

handle	DMA handle pointer.

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

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

Parameters

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

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

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

Parameters

handle	DMA handle pointer.
callback	DMA callback function pointer.

userData	Parameter for callback function.

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

Parameters

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

Note

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

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

This function used to prepare channel transfer configurations.

Parameters

config	Pointer to DMA channel transfer configuration structure.
srcStartAddr	source start address.

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

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

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

Parameters

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

Return values

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

14.7.28 void DMA_SubmitChannelTransferParameter (dma_handle_t * *handle*, uint32_t *xfercfg*, void * *srcStartAddr*, void * *dstStartAddr*, void * *nextDesc*)

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

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

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

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

```
   //define link descriptor table in application with macro
   DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[3]);

   DMA_SetupDescriptor(nextDesc0,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
   DMA_SetupDescriptor(nextDesc1,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
   DMA_SetupDescriptor(nextDesc2,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
   DMA_SetChannelConfig(base, channel, trigger, isPeriph);
   DMA_CreateHandle(handle, base, channel)
   DMA_SubmitChannelTransferParameter(handle,
      DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width, srcInc, dstInc,
bytes), srcStartAddr, dstStartAddr, nextDesc0);
   DMA_StartTransfer(handle);
*
```

Parameters

handle	Pointer to DMA handle.
xferCfg	xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
srcStartAddr	source start address.
dstStartAddr	destination start address.
nextDesc	address of next descriptor.

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

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

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

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

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

Parameters

handle	Pointer to DMA handle.
descriptor	descriptor to submit.

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

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

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

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

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

```
   //define link descriptor table in application with macro
   DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);
   DMA_SetupDescriptor(nextDesc0,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
   DMA_SetupDescriptor(nextDesc1,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
   DMA_SetupDescriptor(nextDesc2,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
   DMA_CreateHandle(handle, base, channel)
   DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,
      trigger,nextDesc0);
   DMA_SubmitChannelTransfer(handle, config)
   DMA_StartTransfer(handle)
*
```

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

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

   DMA_SetupDescriptor(nextDesc0,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
   DMA_SetupDescriptor(nextDesc1,  DMA_CHANNEL_XFER(reload, clrTrig,
      intA, intB, width, srcInc, dstInc, bytes),
```

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

Parameters

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

Return values

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

14.7.31 void DMA_StartTransfer (dma_handle_t * *handle*)

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

Parameters

handle	DMA handle pointer.

14.7.32 void DMA_IRQHandle (DMA_Type * *base*)

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

Parameters

base	DMA base address.

Chapter 15
GPIO: General Purpose I/O

15.1 Overview

The MCUXpresso SDK provides a peripheral driver for the General Purpose I/O (GPIO) module of MCUXpresso SDK devices.

15.2 Function groups

15.2.1 Initialization and deinitialization

The function GPIO_PinInit() initializes the GPIO with specified configuration.

15.2.2 Pin manipulation

The function GPIO_PinWrite() set output state of selected GPIO pin. The function GPIO_PinRead() read input value of selected GPIO pin.

15.2.3 Port manipulation

The function GPIO_PortSet() sets the output level of selected GPIO pins to the logic 1. The function GPIO_PortClear() sets the output level of selected GPIO pins to the logic 0. The function GPIO_PortToggle() reverse the output level of selected GPIO pins. The function GPIO_PortRead() read input value of selected port.

15.2.4 Port masking

The function GPIO_PortMaskedSet() set port mask, only pins masked by 0 will be enabled in following functions. The function GPIO_PortMaskedWrite() sets the state of selected GPIO port, only pins masked by 0 will be affected. The function GPIO_PortMaskedRead() reads the state of selected GPIO port, only pins masked by 0 are enabled for read, pins masked by 1 are read as 0.

15.3 Typical use case

Example use of GPIO API. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/gpio

Files

- file fsl_gpio.h

Data Structures

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

Enumerations

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

Functions

- static void GPIO_PortSet (GPIO_Type ∗base, uint32_t port, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 1.
- static void GPIO_PortClear (GPIO_Type ∗base, uint32_t port, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 0.
- static void GPIO_PortToggle (GPIO_Type ∗base, uint32_t port, uint32_t mask)
  Reverses current output logic of the multiple GPIO pins.

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))
  LPC GPIO driver version 2.1.3.

GPIO Configuration

- void GPIO_PortInit (GPIO_Type ∗base, uint32_t port)
  Initializes the GPIO peripheral.
- void GPIO_PinInit (GPIO_Type ∗base, uint32_t port, uint32_t pin, const gpio_pin_config_t ∗config)
  Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void GPIO_PinWrite (GPIO_Type ∗base, uint32_t port, uint32_t pin, uint8_t output)
  Sets the output level of the one GPIO pin to the logic 1 or 0.

GPIO Input Operations

- static uint32_t GPIO_PinRead (GPIO_Type ∗base, uint32_t port, uint32_t pin)
  Reads the current input value of the GPIO PIN.

15.4 Data Structure Documentation

15.4.1 struct gpio_pin_config_t

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

Data Fields

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

15.5 Macro Definition Documentation

15.5.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))

15.6 Enumeration Type Documentation

15.6.1 enum gpio_pin_direction_t

Enumerator

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

15.7 Function Documentation

15.7.1 void GPIO_PortInit (GPIO_Type * *base*, uint32_t *port*)

This function ungates the GPIO clock.

Parameters

base	GPIO peripheral base pointer.
port	GPIO port number.

15.7.2 void GPIO_PinInit (GPIO_Type * *base*, uint32_t *port*, uint32_t *pin*, const gpio_pin_config_t * *config*)

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

This is an example to define an input pin or output pin configuration:

```
* // Define a digital input pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalInput,
*   0,
* }
* //Define a digital output pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalOutput,
*   0,
* }
*
```

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
pin	GPIO pin number
config	GPIO pin configuration pointer

15.7.3 static void GPIO_PinWrite (GPIO_Type ∗ *base,* uint32_t *port,* uint32_t *pin,* uint8_t *output*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
pin	GPIO pin number
output	GPIO pin output logic level. • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

15.7.4 static uint32_t GPIO_PinRead (GPIO_Type ∗ *base,* uint32_t *port,* uint32_t *pin*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
pin	GPIO pin number

Return values

GPIO	port input value • 0: corresponding pin input low-logic level. • 1: corresponding pin input high-logic level.

15.7.5 static void GPIO_PortSet (GPIO_Type ∗ *base*, uint32_t *port*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

15.7.6 static void GPIO_PortClear (GPIO_Type ∗ *base*, uint32_t *port*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

15.7.7 static void GPIO_PortToggle (GPIO_Type ∗ *base*, uint32_t *port*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

Chapter 16
PINT: Pin Interrupt and Pattern Match Driver

16.1 Overview

The MCUXpresso SDK provides a driver for the Pin Interrupt and Pattern match (PINT).

It can configure one or more pins to generate a pin interrupt when the pin or pattern match conditions are met. The pins do not have to be configured as gpio pins however they must be connected to PINT via INPUTMUX. Only the pin interrupt or pattern match function can be active for interrupt generation. If the pin interrupt function is enabled then the pattern match function can be used for wakeup via RXEV.

16.2 Pin Interrupt and Pattern match Driver operation

PINT_PinInterruptConfig() function configures the pins for pin interrupt.

PINT_PatternMatchConfig() function configures the pins for pattern match.

16.2.1 Pin Interrupt use case

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

16.2.2 Pattern match use case

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

Files

- file fsl_pint.h

Typedefs

- typedef void(∗ pint_cb_t)(pint_pin_int_t pintr, uint32_t pmatch_status)
  PINT Callback function.

Enumerations

- enum pint_pin_enable_t {
  kPINT_PinIntEnableNone = 0U,
  kPINT_PinIntEnableRiseEdge = PINT_PIN_RISE_EDGE,
  kPINT_PinIntEnableFallEdge = PINT_PIN_FALL_EDGE,
  kPINT_PinIntEnableBothEdges = PINT_PIN_BOTH_EDGE,
  kPINT_PinIntEnableLowLevel = PINT_PIN_LOW_LEVEL,
  kPINT_PinIntEnableHighLevel = PINT_PIN_HIGH_LEVEL }

PINT Pin Interrupt enable type.
- enum pint_pin_int_t { kPINT_PinInt0 = 0U }
  PINT Pin Interrupt type.
- enum pint_pmatch_input_src_t {
  kPINT_PatternMatchInp0Src = 0U,
  kPINT_PatternMatchInp1Src = 1U,
  kPINT_PatternMatchInp2Src = 2U,
  kPINT_PatternMatchInp3Src = 3U,
  kPINT_PatternMatchInp4Src = 4U,
  kPINT_PatternMatchInp5Src = 5U,
  kPINT_PatternMatchInp6Src = 6U,
  kPINT_PatternMatchInp7Src = 7U }
  PINT Pattern Match bit slice input source type.
- enum pint_pmatch_bslice_t { kPINT_PatternMatchBSlice0 = 0U }
  PINT Pattern Match bit slice type.
- enum pint_pmatch_bslice_cfg_t {
  kPINT_PatternMatchAlways = 0U,
  kPINT_PatternMatchStickyRise = 1U,
  kPINT_PatternMatchStickyFall = 2U,
  kPINT_PatternMatchStickyBothEdges = 3U,
  kPINT_PatternMatchHigh = 4U,
  kPINT_PatternMatchLow = 5U,
  kPINT_PatternMatchNever = 6U,
  kPINT_PatternMatchBothEdges = 7U }
  PINT Pattern Match configuration type.

Functions

- void PINT_Init (PINT_Type ∗base)
  Initialize PINT peripheral.
- void PINT_PinInterruptConfig (PINT_Type ∗base, pint_pin_int_t intr, pint_pin_enable_t enable, pint_cb_t callback)
  Configure PINT peripheral pin interrupt.
- void PINT_PinInterruptGetConfig (PINT_Type ∗base, pint_pin_int_t pintr, pint_pin_enable_t ∗enable, pint_cb_t ∗callback)
  Get PINT peripheral pin interrupt configuration.
- void PINT_PinInterruptClrStatus (PINT_Type ∗base, pint_pin_int_t pintr)
  Clear Selected pin interrupt status only when the pin was triggered by edge-sensitive.
- static uint32_t PINT_PinInterruptGetStatus (PINT_Type ∗base, pint_pin_int_t pintr)
  Get Selected pin interrupt status.
- void PINT_PinInterruptClrStatusAll (PINT_Type ∗base)
  Clear all pin interrupts status only when pins were triggered by edge-sensitive.
- static uint32_t PINT_PinInterruptGetStatusAll (PINT_Type ∗base)
  Get all pin interrupts status.
- static void PINT_PinInterruptClrFallFlag (PINT_Type ∗base, pint_pin_int_t pintr)
  Clear Selected pin interrupt fall flag.
- static uint32_t PINT_PinInterruptGetFallFlag (PINT_Type ∗base, pint_pin_int_t pintr)
  Get selected pin interrupt fall flag.
- static void PINT_PinInterruptClrFallFlagAll (PINT_Type ∗base)

Clear all pin interrupt fall flags.
- static uint32_t PINT_PinInterruptGetFallFlagAll (PINT_Type ∗base)
  Get all pin interrupt fall flags.
- static void PINT_PinInterruptClrRiseFlag (PINT_Type ∗base, pint_pin_int_t pintr)
  Clear Selected pin interrupt rise flag.
- static uint32_t PINT_PinInterruptGetRiseFlag (PINT_Type ∗base, pint_pin_int_t pintr)
  Get selected pin interrupt rise flag.
- static void PINT_PinInterruptClrRiseFlagAll (PINT_Type ∗base)
  Clear all pin interrupt rise flags.
- static uint32_t PINT_PinInterruptGetRiseFlagAll (PINT_Type ∗base)
  Get all pin interrupt rise flags.
- void PINT_PatternMatchConfig (PINT_Type ∗base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg-_t ∗cfg)
  Configure PINT pattern match.
- void PINT_PatternMatchGetConfig (PINT_Type ∗base, pint_pmatch_bslice_t bslice, pint_pmatch-_cfg_t ∗cfg)
  Get PINT pattern match configuration.
- static uint32_t PINT_PatternMatchGetStatus (PINT_Type ∗base, pint_pmatch_bslice_t bslice)
  Get pattern match bit slice status.
- static uint32_t PINT_PatternMatchGetStatusAll (PINT_Type ∗base)
  Get status of all pattern match bit slices.
- uint32_t PINT_PatternMatchResetDetectLogic (PINT_Type ∗base)
  Reset pattern match detection logic.
- static void PINT_PatternMatchEnable (PINT_Type ∗base)
  Enable pattern match function.
- static void PINT_PatternMatchDisable (PINT_Type ∗base)
  Disable pattern match function.
- static void PINT_PatternMatchEnableRXEV (PINT_Type ∗base)
  Enable RXEV output.
- static void PINT_PatternMatchDisableRXEV (PINT_Type ∗base)
  Disable RXEV output.
- void PINT_EnableCallback (PINT_Type ∗base)
  Enable callback.
- void PINT_DisableCallback (PINT_Type ∗base)
  Disable callback.
- void PINT_Deinit (PINT_Type ∗base)
  Deinitialize PINT peripheral.
- void PINT_EnableCallbackByIndex (PINT_Type ∗base, pint_pin_int_t pintIdx)
  enable callback by pin index.
- void PINT_DisableCallbackByIndex (PINT_Type ∗base, pint_pin_int_t pintIdx)
  disable callback by pin index.

Driver version

- #define FSL_PINT_DRIVER_VERSION (MAKE_VERSION(2, 1, 3))
  Version 2.1.3.

16.3 Typedef Documentation

16.3.1 typedef void(∗ pint_cb_t)(pint_pin_int_t pintr, uint32_t pmatch_status)

16.4 Enumeration Type Documentation

16.4.1 enum pint_pin_enable_t

Enumerator

kPINT_PinIntEnableNone Do not generate Pin Interrupt.
kPINT_PinIntEnableRiseEdge Generate Pin Interrupt on rising edge.
kPINT_PinIntEnableFallEdge Generate Pin Interrupt on falling edge.
kPINT_PinIntEnableBothEdges Generate Pin Interrupt on both edges.
kPINT_PinIntEnableLowLevel Generate Pin Interrupt on low level.
kPINT_PinIntEnableHighLevel Generate Pin Interrupt on high level.

16.4.2 enum pint_pin_int_t

Enumerator

kPINT_PinInt0 Pin Interrupt 0.

16.4.3 enum pint_pmatch_input_src_t

Enumerator

kPINT_PatternMatchInp0Src Input source 0.
kPINT_PatternMatchInp1Src Input source 1.
kPINT_PatternMatchInp2Src Input source 2.
kPINT_PatternMatchInp3Src Input source 3.
kPINT_PatternMatchInp4Src Input source 4.
kPINT_PatternMatchInp5Src Input source 5.
kPINT_PatternMatchInp6Src Input source 6.
kPINT_PatternMatchInp7Src Input source 7.

16.4.4 enum pint_pmatch_bslice_t

Enumerator

kPINT_PatternMatchBSlice0 Bit slice 0.

16.4.5 enum pint_pmatch_bslice_cfg_t

Enumerator

kPINT_PatternMatchAlways Always Contributes to product term match.
kPINT_PatternMatchStickyRise Sticky Rising edge.
kPINT_PatternMatchStickyFall Sticky Falling edge.
kPINT_PatternMatchStickyBothEdges Sticky Rising or Falling edge.
kPINT_PatternMatchHigh High level.
kPINT_PatternMatchLow Low level.
kPINT_PatternMatchNever Never contributes to product term match.
kPINT_PatternMatchBothEdges Either rising or falling edge.

16.5 Function Documentation

16.5.1 void PINT_Init (PINT_Type * *base*)

This function initializes the PINT peripheral and enables the clock.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.2 void PINT_PinInterruptConfig (PINT_Type * *base,* pint_pin_int_t *intr,* pint_pin_enable_t *enable,* pint_cb_t *callback*)

This function configures a given pin interrupt.

Parameters

base	Base address of the PINT peripheral.
intr	Pin interrupt.
enable	Selects detection logic.
callback	Callback.

Return values

None.	

16.5.3 void PINT_PinInterruptGetConfig (PINT_Type * *base*, pint_pin_int_t *pintr*, pint_pin_enable_t * *enable*, pint_cb_t * *callback*)

This function returns the configuration of a given pin interrupt.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.
enable	Pointer to store the detection logic.
callback	Callback.

Return values

None.	

16.5.4 void PINT_PinInterruptClrStatus (PINT_Type * *base*, pint_pin_int_t *pintr*)

This function clears the selected pin interrupt status.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

16.5.5 static uint32_t PINT_PinInterruptGetStatus (PINT_Type * *base*, pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt status.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

status	= 0 No pin interrupt request. = 1 Selected Pin interrupt request active.

16.5.6 void PINT_PinInterruptClrStatusAll (PINT_Type * *base*)

This function clears the status of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.7 static uint32_t PINT_PinInterruptGetStatusAll (PINT_Type * *base*) [inline], [static]

This function returns the status of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

status	Each bit position indicates the status of corresponding pin interrupt. = 0 No pin interrupt request. = 1 Pin interrupt request active.

16.5.8 static void PINT_PinInterruptClrFallFlag (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function clears the selected pin interrupt fall flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

16.5.9 static uint32_t PINT_PinInterruptGetFallFlag (PINT_Type * *base*, pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt fall flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

flag	= 0 Falling edge has not been detected. = 1 Falling edge has been detected.

16.5.10 static void PINT_PinInterruptClrFallFlagAll (PINT_Type * *base*) [inline], [static]

This function clears the fall flag for all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.11 static uint32_t PINT_PinInterruptGetFallFlagAll (PINT_Type * *base*) [inline], [static]

This function returns the fall flag of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

flags	Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

16.5.12 static void PINT_PinInterruptClrRiseFlag (PINT_Type * *base*, pint_pin_int_t *pintr*) [inline], [static]

This function clears the selected pin interrupt rise flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

16.5.13 static uint32_t PINT_PinInterruptGetRiseFlag (PINT_Type * *base*, pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt rise flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

flag	= 0 Rising edge has not been detected. = 1 Rising edge has been detected.

16.5.14 static void PINT_PinInterruptClrRiseFlagAll (PINT_Type * *base*) [inline], [static]

This function clears the rise flag for all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.15 static uint32_t PINT_PinInterruptGetRiseFlagAll (PINT_Type * *base*) [inline], [static]

This function returns the rise flag of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

flags	Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

16.5.16 void PINT_PatternMatchConfig (PINT_Type * *base,* pint_pmatch_bslice_t *bslice,* pint_pmatch_cfg_t * *cfg*)

This function configures a given pattern match bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.
cfg	Pointer to bit slice configuration.

Return values

None.	

16.5.17 void PINT_PatternMatchGetConfig (PINT_Type * *base*, pint_pmatch_bslice_t *bslice*, pint_pmatch_cfg_t * *cfg*)

This function returns the configuration of a given pattern match bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.
cfg	Pointer to bit slice configuration.

Return values

None.	

16.5.18 static uint32_t PINT_PatternMatchGetStatus (PINT_Type * *base*, pint_pmatch_bslice_t *bslice*) [inline], [static]

This function returns the status of selected bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.

Return values

status	= 0 Match has not been detected. = 1 Match has been detected.

16.5.19 static uint32_t PINT_PatternMatchGetStatusAll (PINT_Type * *base*) [inline], [static]

This function returns the status of all bit slices.

Parameters

base	Base address of the PINT peripheral.

Return values

status	Each bit position indicates the match status of corresponding bit slice. = 0 Match has not been detected. = 1 Match has been detected.

16.5.20 uint32_t PINT_PatternMatchResetDetectLogic (PINT_Type * *base*)

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

base	Base address of the PINT peripheral.

Return values

pmstatus	Each bit position indicates the match status of corresponding bit slice. = 0 Match was detected. = 1 Match was not detected.

16.5.21 static void PINT_PatternMatchEnable (PINT_Type * *base*) [inline], [static]

This function enables the pattern match function.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.22 static void PINT_PatternMatchDisable (PINT_Type ∗ *base*) [inline], [static]

This function disables the pattern match function.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.23 static void PINT_PatternMatchEnableRXEV (PINT_Type ∗ *base*) [inline], [static]

This function enables the pattern match RXEV output.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.24 static void PINT_PatternMatchDisableRXEV (PINT_Type ∗ *base*) [inline], [static]

This function disables the pattern match RXEV output.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.25 void PINT_EnableCallback (PINT_Type * *base*)

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

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.26 void PINT_DisableCallback (PINT_Type * *base*)

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

Parameters

base	Base address of the peripheral.

Return values

None.	

16.5.27 void PINT_Deinit (PINT_Type * *base*)

This function disables the PINT clock.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

16.5.28 void PINT_EnableCallbackByIndex (PINT_Type * *base*, pint_pin_int_t *pintIdx*)

This function enables callback by pin index instead of enabling all pins.

Parameters

base	Base address of the peripheral.
pinIdx	pin index.

Return values

None.	

16.5.29 void PINT_DisableCallbackByIndex (PINT_Type * *base*, pint_pin_int_t *pintIdx*)

This function disables callback by pin index instead of disabling all pins.

Parameters

base	Base address of the peripheral.
pinIdx	pin index.

Return values

None.	

Chapter 17
SYSCON: System Configuration

17.1 Overview

The MCUXpresso SDK provides a peripheral clock and power driver for the SYSCON module of MCUXpresso SDK devices. For furter details, see the corresponding chapter.

Files

- file fsl_syscon.h
- file fsl_syscon.h

Functions

- void SYSCON_AttachSignal (SYSCON_Type ∗base, uint32_t index, syscon_connection_t connection)
  Attaches a signal.

Driver version

- #define FSL_SYSON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Group syscon driver version for SDK.

Syscon multiplexing connections

- enum syscon_connection_t { kSYSCON_GpioPort0Pin0ToPintsel = 0U + (PINTSEL_ID << SYSCON_SHIFT) }
  SYSCON connections type.
- #define PINTSEL_ID 0x178U
  Periphinmux IDs.
- #define **SYSCON_SHIFT** 20U

17.2 Macro Definition Documentation

17.2.1 #define FSL_SYSON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

17.3 Enumeration Type Documentation

17.3.1 enum syscon_connection_t

Enumerator

kSYSCON_GpioPort0Pin0ToPintsel Pin Interrupt.

17.4 Function Documentation

17.4.1 void SYSCON_AttachSignal (SYSCON_Type * *base*, uint32_t *index*, syscon_connection_t *connection*)

This function gates the SYSCON clock.

Parameters

base	Base address of the SYSCON peripheral.
index	Destination peripheral to attach the signal to.
connection	Selects connection.

Return values

None.	

Chapter 18
Clock Driver

18.1 Overview

The MCUXpresso SDK provides a peripheral clock driver for the SYSCON module of MCUXpresso SDK devices.

18.2 Function description

Clock driver provides these functions:

- Functions to initialize the Core clock to given frequency
- Functions to configure the clock selection muxes.
- Functions to setup peripheral clock dividers
- Functions to set the flash wait states for the input freuqency
- Functions to get the frequency of the selected clock
- Functions to set PLL frequency

18.2.1 SYSCON Clock frequency functions

SYSCON clock module provides clocks, such as MCLKCLK, ADCCLK, DMICCLK, MCGFLLCLK, FXCOMCLK, WDTOSC, RTCOSC, USBCLK, and SYSPLL. The functions CLOCK_EnableClock() and CLOCK_DisableClock() enables and disables the various clocks. CLOCK_SetupFROClocking() initializes the FRO to 12 MHz, 48 MHz, or 96 MHz frequency. CLOCK_SetupPLLData(), CLOCK_-SetupSystemPLLPrec(), and CLOCK_SetPLLFreq() functions are used to setup the PLL. The SYSCON clock driver provides functions to get the frequency of these clocks, such as CLOCK_GetFreq(), CLOC-K_GetFro12MFreq(), CLOCK_GetExtClkFreq(), CLOCK_GetWdtOscFreq(), CLOCK_GetFroHfFreq(), CLOCK_GetPllOutFreq(), CLOCK_GetOsc32KFreq(), CLOCK_GetCoreSysClkFreq(), CLOCK_GetI2-SMClkFreq(), CLOCK_GetFlexCommClkFreq, and CLOCK_GetAsyncApbClkFreq.

18.2.2 SYSCON clock Selection Muxes

The SYSCON clock driver provides the function to configure the clock selected. The function CLOCK_-AttachClk() is implemented for this. The function selects the clock source for a particular peripheral like MAINCLK, DMIC, FLEXCOMM, USB, ADC, and PLL.

18.2.3 SYSCON clock dividers

The SYSCON clock module provides the function to setup the peripheral clock dividers. The function CLOCK_SetClkDiv() configures the CLKDIV registers for various periperals like USB, DMIC, I2S, SYSTICK, AHB, ADC, and also CLKOUT and TRACE functions.

18.2.4 SYSCON flash wait states

The SYSCON clock driver provides the function CLOCK_SetFLASHAccessCyclesForFreq() that configures FLASHCFG register with a selected FLASHTIM value.

18.3 Typical use case

```
POWER_DisablePD(kPDRUNCFG_PD_FRO_EN); /*!< Ensure FRO is on so that we can switch to its 12MH
```

Files

- file fsl_clock.h

Data Structures

- struct clock_sys_pll_t
  PLL configuration structure. More...

Macros

- #define CLOCK_FRO_SETTING_API_ROM_ADDRESS (0x0F0026F5U)
  FRO clock setting API address in ROM.
- #define CLOCK_FAIM_BASE (0x50010000U)
  FAIM base address.
- #define ADC_CLOCKS
  Clock ip name array for ADC.
- #define ACMP_CLOCKS
  Clock ip name array for ACMP.
- #define DAC_CLOCKS
  Clock ip name array for DAC.
- #define SWM_CLOCKS
  Clock ip name array for SWM.
- #define ROM_CLOCKS
  Clock ip name array for ROM.
- #define SRAM_CLOCKS
  Clock ip name array for SRAM.
- #define IOCON_CLOCKS
  Clock ip name array for IOCON.
- #define GPIO_CLOCKS
  Clock ip name array for GPIO.
- #define GPIO_INT_CLOCKS
  Clock ip name array for GPIO_INT.

- #define DMA_CLOCKS
  Clock ip name array for DMA.
- #define CRC_CLOCKS
  Clock ip name array for CRC.
- #define WWDT_CLOCKS
  Clock ip name array for WWDT.
- #define SCT_CLOCKS
  Clock ip name array for SCT0.
- #define I2C_CLOCKS
  Clock ip name array for I2C.
- #define USART_CLOCKS
  Clock ip name array for I2C.
- #define SPI_CLOCKS
  Clock ip name array for SPI.
- #define CAPT_CLOCKS
  Clock ip name array for CAPT.
- #define CTIMER_CLOCKS
  Clock ip name array for CTIMER.
- #define MTB_CLOCKS
  Clock ip name array for MTB.
- #define MRT_CLOCKS
  Clock ip name array for MRT.
- #define WKT_CLOCKS
  Clock ip name array for WKT.
- #define CLK_GATE_DEFINE(reg, bit) ((((reg)&0xFFU) << 8U) | ((bit)&0xFFU))
  Internal used Clock definition only.

Enumerations

- enum clock_ip_name_t
  Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.
- enum clock_name_t {
  kCLOCK_CoreSysClk,
  kCLOCK_MainClk,
  kCLOCK_Fro,
  kCLOCK_FroDiv,
  kCLOCK_ExtClk,
  kCLOCK_PllOut,
  kCLOCK_WdtOsc,
  kCLOCK_Frg0,
  kCLOCK_Frg1 }
  Clock name used to get clock frequency.
- enum clock_select_t
  Clock Mux Switches CLK_MUX_DEFINE(reg, mux) reg is used to define the mux register mux is used to define the mux value.
- enum clock_divider_t
  Clock divider.
- enum clock_wdt_analog_freq_t
  watch dog analog output frequency
- enum clock_fro_src_t {

kCLOCK_FroSrcLpwrBootValue = 0U,
kCLOCK_FroSrcFroOsc = 1U << SYSCON_FROOSCCTRL_FRO_DIRECT_SHIFT }
fro output frequency source definition
- enum clock_fro_osc_freq_t {
kCLOCK_FroOscOut18M = 18000U,
kCLOCK_FroOscOut24M = 24000U,
kCLOCK_FroOscOut30M = 30000U }
fro oscillator output frequency value definition
- enum clock_sys_pll_src {
kCLOCK_SysPllSrcFRO = 0U,
kCLOCK_SysPllSrcExtClk = 1U,
kCLOCK_SysPllSrcWdtOsc = 2U,
kCLOCK_SysPllSrcFroDiv = 3U }
PLL clock definition.
- enum clock_main_clk_src_t {
kCLOCK_MainClkSrcFro = CLK_MAIN_CLK_MUX_DEFINE(0U, 0U),
kCLOCK_MainClkSrcExtClk = CLK_MAIN_CLK_MUX_DEFINE(1U, 0U),
kCLOCK_MainClkSrcWdtOsc = CLK_MAIN_CLK_MUX_DEFINE(2U, 0U),
kCLOCK_MainClkSrcFroDiv = CLK_MAIN_CLK_MUX_DEFINE(3U, 0U),
kCLOCK_MainClkSrcSysPll = CLK_MAIN_CLK_MUX_DEFINE(0U, 1U) }

Variables

- volatile uint32_t g_Wdt_Osc_Freq
  watchdog oscilltor clock frequency.
- volatile uint32_t g_Ext_Clk_Freq
  external clock frequency.

Driver version

- #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  CLOCK driver version 2.1.0.

Clock gate, mux, and divider.

- static void **CLOCK_EnableClock** (clock_ip_name_t clk)
- static void **CLOCK_DisableClock** (clock_ip_name_t clk)
- static void **CLOCK_Select** (clock_select_t sel)
- static void **CLOCK_SetClkDivider** (clock_divider_t name, uint32_t value)
- static uint32_t **CLOCK_GetClkDivider** (clock_divider_t name)
- static void **CLOCK_SetCoreSysClkDiv** (uint32_t value)
- void CLOCK_SetMainClkSrc (clock_main_clk_src_t src)
  Set main clock reference source.
- void CLOCK_SetFroOutClkSrc (clock_fro_src_t src)
  Set FRO clock source.
- static void **CLOCK_SetFRGClkMul** (uint32_t *base, uint32_t mul)

Get frequency

- uint32_t CLOCK_GetFRG0ClkFreq (void)

Return Frequency of FRG0 Clock.
- uint32_t CLOCK_GetFRG1ClkFreq (void)
  Return Frequency of FRG1 Clock.
- uint32_t CLOCK_GetMainClkFreq (void)
  Return Frequency of Main Clock.
- uint32_t CLOCK_GetFroFreq (void)
  Return Frequency of FRO.
- static uint32_t CLOCK_GetCoreSysClkFreq (void)
  Return Frequency of core.
- uint32_t CLOCK_GetClockOutClkFreq (void)
  Return Frequency of ClockOut.
- uint32_t CLOCK_GetFreq (clock_name_t clockName)
  Return Frequency of selected clock.
- uint32_t CLOCK_GetSystemPLLInClockRate (void)
  Return System PLL input clock rate.
- static uint32_t CLOCK_GetSystemPLLFreq (void)
  Return Frequency of System PLL.
- static uint32_t CLOCK_GetWdtOscFreq (void)
  Get watch dog OSC frequency.
- static uint32_t CLOCK_GetExtClkFreq (void)
  Get external clock frequency.

PLL operations

- void CLOCK_InitSystemPll (const clock_sys_pll_t *config)
  System PLL initialize.
- static void CLOCK_DenitSystemPll (void)
  System PLL Deinitialize.

Fractional clock operations

- bool CLOCK_SetFRG0ClkFreq (uint32_t freq)
  Set FRG0 output frequency.
- bool CLOCK_SetFRG1ClkFreq (uint32_t freq)
  Set FRG1 output frequency.

External/internal oscillator clock operations

- void CLOCK_InitExtClkin (uint32_t clkInFreq)
  Init external CLK IN, select the CLKIN as the external clock source.
- void CLOCK_InitSysOsc (uint32_t oscFreq)
  Init SYS OSC.
- void CLOCK_InitXtalin (uint32_t xtalInFreq)
  XTALIN init function system oscillator is bypassed, sys_osc_clk is fed driectly from the XTALIN.
- static void CLOCK_DeinitSysOsc (void)
  Deinit SYS OSC.
- void CLOCK_InitWdtOsc (clock_wdt_analog_freq_t wdtOscFreq, uint32_t wdtOscDiv)
  Init watch dog OSC Any setting of the FREQSEL bits will yield a Fclkana value within 40% of the listed frequency value.
- static void CLOCK_DeinitWdtOsc (void)
  Deinit watch dog OSC.

- static void CLOCK_SetFroOscFreq (clock_fro_osc_freq_t freq)
  Set FRO oscillator output frequency.
- void SDK_DelayAtLeastUs (uint32_t delay_us)
  Delay at least for several microseconds.

18.4 Data Structure Documentation

18.4.1 struct clock_sys_pll_t

Data Fields

- uint32_t targetFreq
  System pll fclk output frequency, the output frequency should be lower than 100MHZ.
- clock_sys_pll_src src
  System pll clock source.

18.5 Macro Definition Documentation

18.5.1 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

18.5.2 #define CLOCK_FRO_SETTING_API_ROM_ADDRESS (0x0F0026F5U)

18.5.3 #define ADC_CLOCKS

Value:

```
{               \
        kCLOCK_Adc, \
    }
```

18.5.4 #define ACMP_CLOCKS

Value:

```
{                \
        kCLOCK_Acmp, \
    }
```

18.5.5 #define DAC_CLOCKS

Value:

```
{                             \
        kCLOCK_Dac0, kCLOCK_Dac1, \
    }
```

18.5.6 #define SWM_CLOCKS

Value:

```
{                \
        kCLOCK_Swm, \
    }
```

18.5.7 #define ROM_CLOCKS

Value:

```
{                \
        kCLOCK_Rom, \
    }
```

18.5.8 #define SRAM_CLOCKS

Value:

```
{                   \
        kCLOCK_Ram0_1, \
    }
```

18.5.9 #define IOCON_CLOCKS

Value:

```
{                  \
        kCLOCK_Iocon, \
    }
```

18.5.10 #define GPIO_CLOCKS

Value:

```
{                                 \
        kCLOCK_Gpio0, kCLOCK_Gpio1, \
    }
```

18.5.11 #define GPIO_INT_CLOCKS

Value:

```
{                   \
        kCLOCK_GpioInt, \
    }
```

18.5.12 #define DMA_CLOCKS

Value:

```
{               \
        kCLOCK_Dma, \
    }
```

18.5.13 #define CRC_CLOCKS

Value:

```
{               \
        kCLOCK_Crc, \
    }
```

18.5.14 #define WWDT_CLOCKS

Value:

```
{                \
        kCLOCK_Wwdt, \
    }
```

18.5.15 #define SCT_CLOCKS

Value:

```
{               \
        kCLOCK_Sct, \
    }
```

18.5.16 #define I2C_CLOCKS

Value:

```
{                                                                    \
        kCLOCK_I2c0, kCLOCK_I2c1, kCLOCK_I2c2, kCLOCK_I2c3, \
    }
```

18.5.17 #define USART_CLOCKS

Value:

```
{                                                                                         \
        kCLOCK_Uart0, kCLOCK_Uart1, kCLOCK_Uart2, kCLOCK_Uart3, kCLOCK_Uart4, \
    }
```

18.5.18 #define SPI_CLOCKS

Value:

```
{                              \
        kCLOCK_Spi0, kCLOCK_Spi1, \
    }
```

18.5.19 #define CAPT_CLOCKS

Value:

```
{                 \
        kCLOCK_Capt, \
    }
```

18.5.20 #define CTIMER_CLOCKS

Value:

```
{                    \
        kCLOCK_Ctimer0, \
    }
```

18.5.21 #define MTB_CLOCKS

Value:

```
{               \
        kCLOCK_Mtb, \
    }
```

18.5.22 #define MRT_CLOCKS

Value:

```
{               \
        kCLOCK_Mrt, \
    }
```

18.5.23 #define WKT_CLOCKS

Value:

```
{               \
        kCLOCK_Wkt, \
    }
```

18.5.24 #define CLK_GATE_DEFINE(*reg*, *bit*) ((((reg)&0xFFU) << 8U) | ((bit)&0xFFU))

18.6 Enumeration Type Documentation

18.6.1 enum clock_ip_name_t

18.6.2 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Cpu/AHB/AHB matrix/Memories,etc.
kCLOCK_MainClk Main clock.
kCLOCK_Fro FRO18/24/30.
kCLOCK_FroDiv FRO div clock.
kCLOCK_ExtClk External Clock.
kCLOCK_PllOut PLL Output.
kCLOCK_WdtOsc Watchdog Oscillator.
kCLOCK_Frg0 fractional rate0
kCLOCK_Frg1 fractional rate1

18.6.3 enum clock_select_t

18.6.4 enum clock_fro_src_t

Enumerator

kCLOCK_FroSrcLpwrBootValue fro source from the fro oscillator divided by low power boot value

kCLOCK_FroSrcFroOsc fre source from the fro oscillator directly

18.6.5 enum clock_fro_osc_freq_t

Enumerator

kCLOCK_FroOscOut18M FRO oscillator output 18M.
kCLOCK_FroOscOut24M FRO oscillator output 24M.
kCLOCK_FroOscOut30M FRO oscillator output 30M.

18.6.6 enum clock_sys_pll_src

Enumerator

kCLOCK_SysPllSrcFRO system pll source from FRO
kCLOCK_SysPllSrcExtClk system pll source from external clock
kCLOCK_SysPllSrcWdtOsc system pll source from watchdog oscillator
kCLOCK_SysPllSrcFroDiv system pll source from FRO divided clock

18.6.7 enum clock_main_clk_src_t

Enumerator

kCLOCK_MainClkSrcFro main clock source from FRO
kCLOCK_MainClkSrcExtClk main clock source from Ext clock
kCLOCK_MainClkSrcWdtOsc main clock source from watchdog oscillator
kCLOCK_MainClkSrcFroDiv main clock source from FRO Div
kCLOCK_MainClkSrcSysPll main clock source from system pll

18.7 Function Documentation

18.7.1 void CLOCK_SetMainClkSrc (clock_main_clk_src_t *src*)

Parameters

src,reference	clock_main_clk_src_t to set the main clock source.

18.7.2 void CLOCK_SetFroOutClkSrc (clock_fro_src_t *src*)

Parameters

src,please	reference _clock_fro_src definition.

18.7.3 uint32_t CLOCK_GetFRG0ClkFreq (void)

Returns

Frequency of FRG0 Clock.

18.7.4 uint32_t CLOCK_GetFRG1ClkFreq (void)

Returns

Frequency of FRG1 Clock.

18.7.5 uint32_t CLOCK_GetMainClkFreq (void)

Returns

Frequency of Main Clock.

18.7.6 uint32_t CLOCK_GetFroFreq (void)

Returns

Frequency of FRO.

18.7.7 static uint32_t CLOCK_GetCoreSysClkFreq (void) [inline], [static]

Returns

Frequency of core.

18.7.8 uint32_t CLOCK_GetClockOutClkFreq (void)

Returns

Frequency of ClockOut

18.7.9 uint32_t CLOCK_GetFreq (clock_name_t *clockName*)

Returns

Frequency of selected clock

18.7.10 uint32_t CLOCK_GetSystemPLLInClockRate (void)

Returns

System PLL input clock rate

18.7.11 static uint32_t CLOCK_GetSystemPLLFreq (void) [inline], [static]

Returns

Frequency of PLL

18.7.12 static uint32_t CLOCK_GetWdtOscFreq (void) [inline], [static]

Return values

watch	dog OSC frequency value.

18.7.13 static uint32_t CLOCK_GetExtClkFreq (void) [inline], [static]

Return values

external	clock frequency value.

18.7.14 void CLOCK_InitSystemPll (const clock_sys_pll_t * *config*)

Parameters

config	System PLL configurations.

18.7.15 static void CLOCK_DenitSystemPll (void) [inline], [static]

18.7.16 bool CLOCK_SetFRG0ClkFreq (uint32_t *freq*)

Parameters

freq,target	output frequency,freq < input and (input / freq) < 2 should be satisfy.

Return values

true	- successfully, false - input argument is invalid.

18.7.17 bool CLOCK_SetFRG1ClkFreq (uint32_t *freq*)

Parameters

freq,target	output frequency,freq < input and (input / freq) < 2 should be satisfy.

Return values

true	- successfully, false - input argument is invalid.

18.7.18 void CLOCK_InitExtClkin (uint32_t *clkInFreq*)

Parameters

clkInFreq	external clock in frequency.

18.7.19 void CLOCK_InitSysOsc (uint32_t *oscFreq*)

Parameters

oscFreq	oscillator frequency value.

18.7.20 void CLOCK_InitXtalin (uint32_t *xtalInFreq*)

Parameters

xtalInFreq	XTALIN frequency value

Returns

Frequency of PLL

18.7.21 static void CLOCK_DeinitSysOsc (void) `[inline]`, `[static]`

Parameters

config	oscillator configuration.

18.7.22 void CLOCK_InitWdtOsc (clock_wdt_analog_freq_t *wdtOscFreq,* uint32_t *wdtOscDiv*)

The watchdog oscillator is the clock source with the lowest power consumption. If accurate timing is required, use the FRO or system oscillator. The frequency of the watchdog oscillator is undefined after reset. The watchdog oscillator frequency must be programmed by writing to the WDTOSCCTRL register before using the watchdog oscillator. Watchdog osc output frequency = wdtOscFreq / wdtOscDiv, should in range 9.3KHZ to 2.3MHZ.

Parameters

wdtOscFreq	watch dog analog part output frequency, reference _wdt_analog_output_freq.
wdtOscDiv	watch dog analog part output frequency divider, shoule be a value >= 2U and multiple of 2

18.7.23 static void CLOCK_DeinitWdtOsc (void) [inline], [static]

Parameters

config	oscillator configuration.

18.7.24 static void CLOCK_SetFroOscFreq (clock_fro_osc_freq_t *freq*) [inline], [static]

Initialize the FRO clock to given frequency (18, 24 or 30 MHz).

Parameters

freq,please	reference clock_fro_osc_freq_t definition, frequency must be one of 18000, 24000 or 30000 KHz.

18.7.25 void SDK_DelayAtLeastUs (uint32_t *delay_us*)

Please note that, this API will calculate the microsecond period with the maximum supported CPU frequency, so this API will only delay for at least the given microseconds, if precise delay count was needed, please implement a new timer count to achieve this function.

Parameters

delay_us	Delay time in unit of microsecond.

18.8 Variable Documentation

18.8.1 volatile uint32_t g_Wdt_Osc_Freq

This variable is used to store the watchdog oscillator frequency which is set by CLOCK_InitWdtOsc, and it is returned by CLOCK_GetWdtOscFreq.

18.8.2 volatile uint32_t g_Ext_Clk_Freq

This variable is used to store the external clock frequency which is include external oscillator clock and external clk in clock frequency value, it is set by CLOCK_InitExtClkin when CLK IN is used as external clock or by CLOCK_InitSysOsc when external oscillator is used as external clock ,and it is returned by CLOCK_GetExtClkFreq.

Chapter 19
I2C: Inter-Integrated Circuit Driver

19.1 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 are feature/property target 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 the knowledge of 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 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 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.

19.2 Typical use case

19.2.1 Master Operation in functional method

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

19.2.2 Master Operation in DMA transactional method

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

19.2.3 Slave Operation in functional method

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

19.2.4 Slave Operation in interrupt transactional method

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

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C Master Driver
- I2C Slave Driver

19.3 I2C Driver

19.3.1 Overview

Files

- file fsl_i2c.h

Macros

- #define I2C_WAIT_TIMEOUT 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/
  Timeout times for waiting flag.
- #define I2C_STAT_MSTCODE_IDLE (0)
  Master Idle State Code.
- #define I2C_STAT_MSTCODE_RXREADY (1)
  Master Receive Ready State Code.
- #define I2C_STAT_MSTCODE_TXREADY (2)
  Master Transmit Ready State Code.
- #define I2C_STAT_MSTCODE_NACKADR (3)
  Master NACK by slave on address State Code.
- #define I2C_STAT_MSTCODE_NACKDAT (4)
  Master NACK by slave on data State Code.

Enumerations

- enum _i2c_status {
  kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_LPC_I2C, 0),
  kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_LPC_I2C, 1),
  kStatus_I2C_Nak = MAKE_STATUS(kStatusGroup_LPC_I2C, 2),
  kStatus_I2C_InvalidParameter,
  kStatus_I2C_BitError = MAKE_STATUS(kStatusGroup_LPC_I2C, 4),
  kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_LPC_I2C, 5),
  kStatus_I2C_NoTransferInProgress,
  kStatus_I2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_LPC_I2C, 7) ,
  kStatus_I2C_Addr_Nak = MAKE_STATUS(kStatusGroup_LPC_I2C, 10),
  kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_LPC_I2C, 11) }
  I2C status return codes.

Driver version

- #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
  I2C driver version 2.0.2.

19.3.2 Macro Definition Documentation

19.3.2.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

19.3.2.2 #define I2C_WAIT_TIMEOUT 0U /* Define to zero means keep waiting until the flag is assert/deassert. */

19.3.3 Enumeration Type Documentation

19.3.3.1 enum _i2c_status

Enumerator

kStatus_I2C_Busy The master is already performing a transfer.
kStatus_I2C_Idle The slave driver is idle.
kStatus_I2C_Nak The slave device sent a NAK in response to a byte.
kStatus_I2C_InvalidParameter Unable to proceed due to invalid parameter.
kStatus_I2C_BitError Transferred bit was not seen on the bus.
kStatus_I2C_ArbitrationLost Arbitration lost error.
kStatus_I2C_NoTransferInProgress Attempt to abort a transfer when one is not in progress.
kStatus_I2C_DmaRequestFail DMA request failed.
kStatus_I2C_Addr_Nak NAK received during the address probe.
kStatus_I2C_Timeout Timeout poling status flags.

19.4 I2C Master Driver

19.4.1 Overview

Data Structures

- struct i2c_master_config_t
  Structure with settings to initialize the I2C master module. More...
- struct i2c_master_transfer_t
  Non-blocking transfer descriptor structure. More...
- struct i2c_master_handle_t
  Driver handle for master non-blocking APIs. More...

Typedefs

- typedef void(∗ i2c_master_transfer_callback_t)(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Master completion callback function pointer type.

Enumerations

- enum _i2c_master_flags {
  kI2C_MasterPendingFlag = I2C_STAT_MSTPENDING_MASK,
  kI2C_MasterArbitrationLostFlag,
  kI2C_MasterStartStopErrorFlag }
  I2C master peripheral flags.
- enum i2c_direction_t {
  kI2C_Write = 0U,
  kI2C_Read = 1U }
  Direction of master and slave transfers.
- enum _i2c_master_transfer_flags {
  kI2C_TransferDefaultFlag = 0x00U,
  kI2C_TransferNoStartFlag = 0x01U,
  kI2C_TransferRepeatedStartFlag = 0x02U,
  kI2C_TransferNoStopFlag = 0x04U }
  Transfer option flags.
- enum _i2c_transfer_states
  States for the state machine used by transactional APIs.

Initialization and deinitialization

- void I2C_MasterGetDefaultConfig (i2c_master_config_t ∗masterConfig)
  Provides a default configuration for the I2C master peripheral.
- void I2C_MasterInit (I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)

Initializes the I2C master peripheral.
- void I2C_MasterDeinit (I2C_Type *base)
  Deinitializes the I2C master peripheral.
- uint32_t I2C_GetInstance (I2C_Type *base)
  Returns an instance number given a base address.
- static void I2C_MasterReset (I2C_Type *base)
  Performs a software reset.
- static void I2C_MasterEnable (I2C_Type *base, bool enable)
  Enables or disables the I2C module as master.

Status

- static uint32_t I2C_GetStatusFlags (I2C_Type *base)
  Gets the I2C status flags.
- static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C master status flag state.

Interrupts

- static void I2C_EnableInterrupts (I2C_Type *base, uint32_t interruptMask)
  Enables the I2C master interrupt requests.
- static void I2C_DisableInterrupts (I2C_Type *base, uint32_t interruptMask)
  Disables the I2C master interrupt requests.
- static uint32_t I2C_GetEnabledInterrupts (I2C_Type *base)
  Returns the set of currently enabled I2C master interrupt requests.

Bus operations

- void I2C_MasterSetBaudRate (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the I2C bus frequency for master transactions.
- static bool I2C_MasterGetBusIdleState (I2C_Type *base)
  Returns whether the bus is idle.
- 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.
- static 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 void *txBuff, size_t txSize, uint32_t flags)
  Performs a polling send transfer on the I2C bus.
- status_t I2C_MasterReadBlocking (I2C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)
  Performs a polling receive transfer 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.

Non-blocking

- void I2C_MasterTransferCreateHandle (I2C_Type ∗base, i2c_master_handle_t ∗handle, i2c_-master_transfer_callback_t callback, void ∗userData)
  Creates a new handle for the I2C master non-blocking APIs.
- status_t I2C_MasterTransferNonBlocking (I2C_Type ∗base, i2c_master_handle_t ∗handle, i2c_-master_transfer_t ∗xfer)
  Performs a non-blocking transaction on the I2C bus.
- status_t I2C_MasterTransferGetCount (I2C_Type ∗base, i2c_master_handle_t ∗handle, size_t ∗count)
  Returns number of bytes transferred so far.
- status_t I2C_MasterTransferAbort (I2C_Type ∗base, i2c_master_handle_t ∗handle)
  Terminates a non-blocking I2C master transmission early.

IRQ handler

- void I2C_MasterTransferHandleIRQ (I2C_Type ∗base, void ∗i2cHandle)
  Reusable routine to handle master interrupts.

19.4.2 Data Structure Documentation

19.4.2.1 struct i2c_master_config_t

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Data Fields

- bool enableMaster
  Whether to enable master mode.
- uint32_t baudRate_Bps
  Desired baud rate in bits per second.
- bool enableTimeout
  Enable internal timeout function.

19.4.2.1.0.12 Field Documentation

19.4.2.1.0.12.1 bool i2c_master_config_t::enableMaster

19.4.2.1.0.12.2 uint32_t i2c_master_config_t::baudRate_Bps

19.4.2.1.0.12.3 bool i2c_master_config_t::enableTimeout

19.4.2.2 struct _i2c_master_transfer

I2C master transfer typedef.

This structure is used to pass transaction parameters to the I2C_MasterTransferNonBlocking() API.

Data Fields

- uint32_t flags
  Bit mask of options for the transfer.
- uint16_t slaveAddress
  The 7-bit slave address.
- i2c_direction_t direction
  Either kI2C_Read or kI2C_Write.
- uint32_t subaddress
  Sub address.
- size_t subaddressSize
  Length of sub address to send in bytes.
- void * data
  Pointer to data to transfer.
- size_t dataSize
  Number of bytes to transfer.

19.4.2.2.0.13 Field Documentation

19.4.2.2.0.13.1 uint32_t i2c_master_transfer_t::flags

See enumeration _i2c_master_transfer_flags for available options. Set to 0 or kI2C_TransferDefaultFlag for normal transfers.

19.4.2.2.0.13.2 uint16_t i2c_master_transfer_t::slaveAddress

19.4.2.2.0.13.3 i2c_direction_t i2c_master_transfer_t::direction

19.4.2.2.0.13.4 uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

19.4.2.2.0.13.5 size_t i2c_master_transfer_t::subaddressSize

Maximum size is 4 bytes.

19.4.2.2.0.13.6 void* i2c_master_transfer_t::data

19.4.2.2.0.13.7 size_t i2c_master_transfer_t::dataSize

19.4.2.3 struct _i2c_master_handle

I2C master handle typedef.

Note

The contents of this structure are private and subject to change.

Data Fields

- uint8_t state
  Transfer state machine current state.
- uint32_t transferCount
  Indicates progress of the transfer.
- uint32_t remainingBytes
  Remaining byte count in current state.
- uint8_t * buf
  Buffer pointer for current state.
- i2c_master_transfer_t transfer
  Copy of the current transfer info.
- i2c_master_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.

19.4.2.3.0.14 Field Documentation

19.4.2.3.0.14.1 uint8_t i2c_master_handle_t::state

19.4.2.3.0.14.2 uint32_t i2c_master_handle_t::remainingBytes

19.4.2.3.0.14.3 uint8_t* i2c_master_handle_t::buf

19.4.2.3.0.14.4 i2c_master_transfer_t i2c_master_handle_t::transfer

19.4.2.3.0.14.5 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

19.4.2.3.0.14.6 void* i2c_master_handle_t::userData

19.4.3 Typedef Documentation

19.4.3.1 typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t completionStatus, void *userData)

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to I2C_MasterTransferCreateHandle().

Parameters

base	The I2C peripheral base address.
completion-Status	Either kStatus_Success or an error code describing how the transfer completed.
userData	Arbitrary pointer-sized value passed from the application.

19.4.4 Enumeration Type Documentation

19.4.4.1 enum _i2c_master_flags

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kI2C_MasterPendingFlag The I2C module is waiting for software interaction.
kI2C_MasterArbitrationLostFlag The arbitration of the bus was lost. There was collision on the bus
kI2C_MasterStartStopErrorFlag There was an error during start or stop phase of the transaction.

19.4.4.2 enum i2c_direction_t

Enumerator

kI2C_Write Master transmit.
kI2C_Read Master receive.

19.4.4.3 enum _i2c_master_transfer_flags

Note

These enumerations are intended to be OR'd together to form a bit mask of options for the _i2c_-master_transfer::flags field.

Enumerator

kI2C_TransferDefaultFlag Transfer starts with a start signal, stops with a stop signal.
kI2C_TransferNoStartFlag Don't send a start condition, address, and sub address.
kI2C_TransferRepeatedStartFlag Send a repeated start condition.
kI2C_TransferNoStopFlag Don't send a stop condition.

19.4.4.4 enum _i2c_transfer_states

19.4.5 Function Documentation

19.4.5.1 void I2C_MasterGetDefaultConfig (i2c_master_config_t * *masterConfig*)

This function provides the following default configuration for the I2C master peripheral:

```
*  masterConfig->enableMaster            = true;
*  masterConfig->baudRate_Bps            = 100000U;
*  masterConfig->enableTimeout           = false;
*
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with I2C_MasterInit().

Parameters

out	*masterConfig*	User provided configuration structure for default values. Refer to i2c_-master_config_t.

19.4.5.2 void I2C_MasterInit (I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

base	The I2C peripheral base address.
masterConfig	User provided peripheral configuration. Use I2C_MasterGetDefaultConfig() to get a set of defaults that you can override.
srcClock_Hz	Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

19.4.5.3 void I2C_MasterDeinit (I2C_Type * *base*)

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

19.4.5.4 uint32_t I2C_GetInstance (I2C_Type * *base*)

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

base	The I2C peripheral base address.

Returns

I2C instance number starting from 0.

19.4.5.5 static void I2C_MasterReset (I2C_Type * *base*) `[inline]`, `[static]`

Restores the I2C master peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

19.4.5.6 static void I2C_MasterEnable (I2C_Type * *base*, bool *enable*) `[inline]`, `[static]`

Parameters

base	The I2C peripheral base address.
enable	Pass true to enable or false to disable the specified I2C as master.

19.4.5.7 static uint32_t I2C_GetStatusFlags (I2C_Type * *base*) `[inline]`, `[static]`

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The I2C peripheral base address.

Returns

State of the status flags:
- 1: related status flag is set.
- 0: related status flag is not set.

See Also

_i2c_master_flags

19.4.5.8 static void I2C_MasterClearStatusFlags (I2C_Type ∗ *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- kI2C_MasterArbitrationLostFlag
- kI2C_MasterStartStopErrorFlag

Attempts to clear other flags has no effect.

Parameters

base	The I2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _i2c_master_flags enumerators OR'd together. You may pass the result of a previous call to I2C_GetStatusFlags().

See Also

_i2c_master_flags.

19.4.5.9 static void I2C_EnableInterrupts (I2C_Type ∗ *base*, uint32_t *interruptMask*) [inline], [static]

Parameters

base	The I2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _i2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

19.4.5.10 static void I2C_DisableInterrupts (I2C_Type ∗ *base,* uint32_t *interruptMask*) [inline], [static]

Parameters

base	The I2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _i2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

19.4.5.11 static uint32_t I2C_GetEnabledInterrupts (I2C_Type ∗ *base*) [inline], [static]

Parameters

base	The I2C peripheral base address.

Returns

A bitmask composed of _i2c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

19.4.5.12 void I2C_MasterSetBaudRate (I2C_Type ∗ *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz*)

The I2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

base	The I2C peripheral base address.
srcClock_Hz	I2C functional clock frequency in Hertz.
baudRate_Bps	Requested bus frequency in bits per second.

19.4.5.13 static bool I2C_MasterGetBusIdleState (I2C_Type * *base*) [inline], [static]

Requires the master mode to be enabled.

Parameters

base	The I2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

19.4.5.14 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.

19.4.5.15 status_t I2C_MasterStop (I2C_Type * *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

19.4.5.16 static status_t I2C_MasterRepeatedStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*) [inline], [static]

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.

19.4.5.17 status_t I2C_MasterWriteBlocking (I2C_Type * *base*, const void * *txBuff*, size_t *txSize*, uint32_t *flags*)

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_I2C_Nak.

Parameters

base	The I2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.
flags	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

kStatus_Success	Data was sent successfully.
kStatus_I2C_Busy	Another master is currently utilizing the bus.
kStatus_I2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_I2C_Arbitration-Lost	Arbitration lost error.

19.4.5.18 status_t I2C_MasterReadBlocking (I2C_Type * *base*, void * *rxBuff*, size_t *rxSize*, uint32_t *flags*)

Parameters

base	The I2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.
flags	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

kStatus_Success	Data was received successfully.
kStatus_I2C_Busy	Another master is currently utilizing the bus.
kStatus_I2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_I2C_Arbitration-Lost	Arbitration lost error.

19.4.5.19 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.

19.4.5.20 void I2C_MasterTransferCreateHandle (I2C_Type ∗ *base*, i2c_master_handle_t ∗ *handle*, i2c_master_transfer_callback_t *callback*, void ∗ *userData*)

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I2C_MasterTransferAbort() API shall be called.

Parameters

	base	The I2C peripheral base address.
out	*handle*	Pointer to the I2C master driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

19.4.5.21 status_t I2C_MasterTransferNonBlocking (I2C_Type ∗ *base*, i2c_master_handle_t ∗ *handle*, i2c_master_transfer_t ∗ *xfer*)

Parameters

base	The I2C peripheral base address.
handle	Pointer to the I2C master driver handle.
xfer	The pointer to the transfer descriptor.

Return values

kStatus_Success	The transaction was started successfully.
kStatus_I2C_Busy	Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

19.4.5.22 status_t I2C_MasterTransferGetCount (I2C_Type * *base,* i2c_master_handle_t * *handle,* size_t * *count*)

Parameters

	base	The I2C peripheral base address.
	handle	Pointer to the I2C master driver handle.
`out`	*count*	Number of bytes transferred so far by the non-blocking transaction.

Return values

kStatus_Success	
kStatus_I2C_Busy	

19.4.5.23 status_t I2C_MasterTransferAbort (I2C_Type * *base*, i2c_master_handle_t * *handle*)

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

Parameters

base	The I2C peripheral base address.
handle	Pointer to the I2C master driver handle.

Return values

kStatus_Success	A transaction was successfully aborted.
kStatus_I2C_Timeout	Abort failure due to flags polling timeout.

19.4.5.24 void I2C_MasterTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Note

This function does not need to be called unless you are reimplementing the nonblocking API's interrupt handler routines to add special functionality.

Parameters

base	The I2C peripheral base address.
handle	Pointer to the I2C master driver handle i2c_master_handle_t.

19.5 I2C Slave Driver

19.5.1 Overview

Data Structures

- struct i2c_slave_address_t
  Data structure with 7-bit Slave address and Slave address disable. More...
- struct i2c_slave_config_t
  Structure with settings to initialize the I2C slave module. More...
- struct i2c_slave_transfer_t
  I2C slave transfer structure. More...
- struct i2c_slave_handle_t
  I2C slave handle structure. More...

Typedefs

- typedef void(∗ i2c_slave_transfer_callback_t)(I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer, void ∗userData)
  Slave event callback function pointer type.
- typedef void(∗ i2c_isr_t)(I2C_Type ∗base, void ∗i2cHandle)
  Typedef for interrupt handler.

Enumerations

- enum _i2c_slave_flags {
  kI2C_SlavePendingFlag = I2C_STAT_SLVPENDING_MASK,
  kI2C_SlaveNotStretching,
  kI2C_SlaveSelected = I2C_STAT_SLVSEL_MASK,
  kI2C_SaveDeselected }
  I2C slave peripheral flags.
- enum i2c_slave_address_register_t {
  kI2C_SlaveAddressRegister0 = 0U,
  kI2C_SlaveAddressRegister1 = 1U,
  kI2C_SlaveAddressRegister2 = 2U,
  kI2C_SlaveAddressRegister3 = 3U }
  I2C slave address register.
- enum i2c_slave_address_qual_mode_t {
  kI2C_QualModeMask = 0U,
  kI2C_QualModeExtend }
  I2C slave address match options.
- enum i2c_slave_bus_speed_t
  I2C slave bus speed options.
- enum i2c_slave_transfer_event_t {

kI2C_SlaveAddressMatchEvent = 0x01U,
kI2C_SlaveTransmitEvent = 0x02U,
kI2C_SlaveReceiveEvent = 0x04U,
kI2C_SlaveCompletionEvent = 0x20U,
kI2C_SlaveDeselectedEvent,
kI2C_SlaveAllEvents }
Set of events sent to the callback for non blocking slave transfers.
- enum i2c_slave_fsm_t
  I2C slave software finite state machine states.

Variables

- i2c_isr_t s_i2cMasterIsr
  Pointer to master IRQ handler for each instance.
- void ∗ s_i2cHandle [FSL_FEATURE_SOC_I2C_COUNT]
  Pointers to i2c handles for each instance.
- const IRQn_Type s_i2cIRQ [ ]
  IRQ name array.

Slave initialization and deinitialization

- void I2C_SlaveGetDefaultConfig (i2c_slave_config_t ∗slaveConfig)
  Provides a default configuration for the I2C slave peripheral.
- status_t I2C_SlaveInit (I2C_Type ∗base, const i2c_slave_config_t ∗slaveConfig, uint32_t srcClock_Hz)
  Initializes the I2C slave peripheral.
- void I2C_SlaveSetAddress (I2C_Type ∗base, i2c_slave_address_register_t addressRegister, uint8_t address, bool addressDisable)
  Configures Slave Address n register.
- void I2C_SlaveDeinit (I2C_Type ∗base)
  Deinitializes the I2C slave peripheral.
- static void I2C_SlaveEnable (I2C_Type ∗base, bool enable)
  Enables or disables the I2C module as slave.

Slave status

- static void I2C_SlaveClearStatusFlags (I2C_Type ∗base, uint32_t statusMask)
  Clears the I2C status flag state.

Slave bus operations

- status_t I2C_SlaveWriteBlocking (I2C_Type ∗base, const uint8_t ∗txBuff, size_t txSize)
  Performs a polling send transfer on the I2C bus.
- status_t I2C_SlaveReadBlocking (I2C_Type ∗base, uint8_t ∗rxBuff, size_t rxSize)
  Performs a polling receive transfer on the I2C bus.

Slave non-blocking

- void I2C_SlaveTransferCreateHandle (I2C_Type ∗base, i2c_slave_handle_t ∗handle, i2c_slave_transfer_callback_t callback, void ∗userData)
  Creates a new handle for the I2C slave non-blocking APIs.
- status_t I2C_SlaveTransferNonBlocking (I2C_Type ∗base, i2c_slave_handle_t ∗handle, uint32_t eventMask)
  Starts accepting slave transfers.
- status_t I2C_SlaveSetSendBuffer (I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer, const void ∗txData, size_t txSize, uint32_t eventMask)
  Starts accepting master read from slave requests.
- status_t I2C_SlaveSetReceiveBuffer (I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer, void ∗rxData, size_t rxSize, uint32_t eventMask)
  Starts accepting master write to slave requests.
- static uint32_t I2C_SlaveGetReceivedAddress (I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer)
  Returns the slave address sent by the I2C master.
- void I2C_SlaveTransferAbort (I2C_Type ∗base, i2c_slave_handle_t ∗handle)
  Aborts the slave non-blocking transfers.
- 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.

Slave IRQ handler

- void I2C_SlaveTransferHandleIRQ (I2C_Type ∗base, void ∗i2cHandle)
  Reusable routine to handle slave interrupts.

19.5.2 Data Structure Documentation

19.5.2.1 struct i2c_slave_address_t

Data Fields

- uint8_t address
  7-bit Slave address SLVADR.
- bool addressDisable
  Slave address disable SADISABLE.

19.5.2.1.0.15 Field Documentation

19.5.2.1.0.15.1 uint8_t i2c_slave_address_t::address

19.5.2.1.0.15.2 bool i2c_slave_address_t::addressDisable

19.5.2.2 struct i2c_slave_config_t

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Data Fields

- i2c_slave_address_t address0
  Slave's 7-bit address and disable.
- i2c_slave_address_t address1
  Alternate slave 7-bit address and disable.
- i2c_slave_address_t address2
  Alternate slave 7-bit address and disable.
- i2c_slave_address_t address3
  Alternate slave 7-bit address and disable.
- i2c_slave_address_qual_mode_t qualMode
  Qualify mode for slave address 0.
- uint8_t qualAddress
  Slave address qualifier for address 0.
- i2c_slave_bus_speed_t busSpeed
  Slave bus speed mode.
- bool enableSlave
  Enable slave mode.

19.5.2.2.0.16 Field Documentation

19.5.2.2.0.16.1 i2c_slave_address_t i2c_slave_config_t::address0

19.5.2.2.0.16.2 i2c_slave_address_t i2c_slave_config_t::address1

19.5.2.2.0.16.3 i2c_slave_address_t i2c_slave_config_t::address2

19.5.2.2.0.16.4 i2c_slave_address_t i2c_slave_config_t::address3

19.5.2.2.0.16.5 i2c_slave_address_qual_mode_t i2c_slave_config_t::qualMode

19.5.2.2.0.16.6 uint8_t i2c_slave_config_t::qualAddress

19.5.2.2.0.16.7 i2c_slave_bus_speed_t i2c_slave_config_t::busSpeed

If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The busSpeed value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the busSpeed mode is unknown at compile time, use the longest data setup time kI2C_SlaveStandardMode (250 ns)

19.5.2.2.0.16.8 bool i2c_slave_config_t::enableSlave

19.5.2.3 struct i2c_slave_transfer_t

Data Fields

- i2c_slave_handle_t * handle
  Pointer to handle that contains this transfer.
- i2c_slave_transfer_event_t event
  Reason the callback is being invoked.
- uint8_t receivedAddress
  Matching address send by master.
- uint32_t eventMask
  Mask of enabled events.
- uint8_t * rxData
  Transfer buffer for receive data.
- const uint8_t * txData
  Transfer buffer for transmit data.
- size_t txSize
  Transfer size.
- size_t rxSize
  Transfer size.
- size_t transferredCount
  Number of bytes transferred during this transfer.
- status_t completionStatus
  Success or error code describing how the transfer completed.

19.5.2.3.0.17 Field Documentation

19.5.2.3.0.17.1 i2c_slave_handle_t* i2c_slave_transfer_t::handle

19.5.2.3.0.17.2 i2c_slave_transfer_event_t i2c_slave_transfer_t::event

19.5.2.3.0.17.3 uint8_t i2c_slave_transfer_t::receivedAddress

7-bits plus R/nW bit0

19.5.2.3.0.17.4 uint32_t i2c_slave_transfer_t::eventMask

19.5.2.3.0.17.5 size_t i2c_slave_transfer_t::transferredCount

19.5.2.3.0.17.6 status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

19.5.2.4 struct _i2c_slave_handle

I2C slave handle typedef.

Note

> The contents of this structure are private and subject to change.

Data Fields

- volatile i2c_slave_transfer_t transfer
  I2C slave transfer.
- volatile bool isBusy
  Whether transfer is busy.
- volatile i2c_slave_fsm_t slaveFsm
  slave transfer state machine.
- i2c_slave_transfer_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback.

19.5.2.4.0.18 Field Documentation

19.5.2.4.0.18.1 volatile i2c_slave_transfer_t i2c_slave_handle_t::transfer

19.5.2.4.0.18.2 volatile bool i2c_slave_handle_t::isBusy

19.5.2.4.0.18.3 volatile i2c_slave_fsm_t i2c_slave_handle_t::slaveFsm

19.5.2.4.0.18.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

19.5.2.4.0.18.5 void∗ i2c_slave_handle_t::userData

19.5.3 Typedef Documentation

19.5.3.1 typedef void(∗ i2c_slave_transfer_callback_t)(I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer, void ∗userData)

This callback is used only for the slave non-blocking transfer API. To install a callback, use the I2C_-SlaveSetCallback() function after you have created a handle.

Parameters

base	Base address for the I2C instance on which the event occurred.
transfer	Pointer to transfer descriptor containing values passed to and/or from the callback.
userData	Arbitrary pointer-sized value passed from the application.

19.5.3.2 typedef void(∗ i2c_isr_t)(I2C_Type ∗base, void ∗i2cHandle)

19.5.4 Enumeration Type Documentation

19.5.4.1 enum _i2c_slave_flags

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kI2C_SlavePendingFlag The I2C module is waiting for software interaction.
kI2C_SlaveNotStretching Indicates whether the slave is currently stretching clock (0 = yes, 1 = no).

kI2C_SlaveSelected Indicates whether the slave is selected by an address match.
kI2C_SaveDeselected Indicates that slave was previously deselected (deselect event took place, w1c).

19.5.4.2 enum i2c_slave_address_register_t

Enumerator

kI2C_SlaveAddressRegister0 Slave Address 0 register.
kI2C_SlaveAddressRegister1 Slave Address 1 register.
kI2C_SlaveAddressRegister2 Slave Address 2 register.
kI2C_SlaveAddressRegister3 Slave Address 3 register.

19.5.4.3 enum i2c_slave_address_qual_mode_t

Enumerator

kI2C_QualModeMask The SLVQUAL0 field (qualAddress) is used as a logical mask for matching address0.
kI2C_QualModeExtend The SLVQUAL0 (qualAddress) field is used to extend address 0 matching in a range of addresses.

19.5.4.4 enum i2c_slave_bus_speed_t

19.5.4.5 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() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

kI2C_SlaveAddressMatchEvent Received the slave address after a start or repeated start.
kI2C_SlaveTransmitEvent Callback is requested to provide data to transmit (slave-transmitter role).

kI2C_SlaveReceiveEvent Callback is requested to provide a buffer in which to place received data (slave-receiver role).
kI2C_SlaveCompletionEvent All data in the active transfer have been consumed.
kI2C_SlaveDeselectedEvent The slave function has become deselected (SLVSEL flag changing from 1 to 0.
kI2C_SlaveAllEvents Bit mask of all available events.

19.5.5 Function Documentation

19.5.5.1 void I2C_SlaveGetDefaultConfig (i2c_slave_config_t ∗ *slaveConfig*)

This function provides the following default configuration for the I2C slave peripheral:

```
*  slaveConfig->enableSlave = true;
*  slaveConfig->address0.disable = false;
*  slaveConfig->address0.address = 0u;
*  slaveConfig->address1.disable = true;
*  slaveConfig->address2.disable = true;
*  slaveConfig->address3.disable = true;
*  slaveConfig->busSpeed = kI2C_SlaveStandardMode;
*
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with I2C_SlaveInit(). Be sure to override at least the *address0.address* member of the configuration structure with the desired slave address.

Parameters

out	*slaveConfig*	User provided configuration structure that is set to default values. Refer to i2c_slave_config_t.

19.5.5.2 status_t I2C_SlaveInit (I2C_Type * *base*, const i2c_slave_config_t * *slaveConfig*, uint32_t *srcClock_Hz*)

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

base	The I2C peripheral base address.
slaveConfig	User provided peripheral configuration. Use I2C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
srcClock_Hz	Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

19.5.5.3 void I2C_SlaveSetAddress (I2C_Type * *base*, i2c_slave_address_register_t *addressRegister*, uint8_t *address*, bool *addressDisable*)

This function writes new value to Slave Address register.

Parameters

base	The I2C peripheral base address.
address-Register	The module supports multiple address registers. The parameter determines which one shall be changed.
address	The slave address to be stored to the address register for matching.
addressDisable	Disable matching of the specified address register.

19.5.5.4 void I2C_SlaveDeinit (I2C_Type * *base*)

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

19.5.5.5 static void I2C_SlaveEnable (I2C_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	The I2C peripheral base address.
enable	True to enable or flase to disable.

19.5.5.6 static void I2C_SlaveClearStatusFlags (I2C_Type * *base,* uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

Parameters

base	The I2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _i2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to I2C_SlaveGetStatusFlags().

See Also

_i2c_slave_flags.

19.5.5.7 status_t I2C_SlaveWriteBlocking (I2C_Type * *base,* const uint8_t * *txBuff,* size_t *txSize*)

The function executes blocking address phase and blocking data phase.

Parameters

base	The I2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been sent.
kStatus_Fail Unexpected slave state (master data write while master read from slave is expected).

19.5.5.8 status_t I2C_SlaveReadBlocking (I2C_Type ∗ *base*, uint8_t ∗ *rxBuff*, size_t *rxSize*)

The function executes blocking address phase and blocking data phase.

Parameters

base	The I2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been received.
kStatus_Fail Unexpected slave state (master data read while master write to slave is expected).

19.5.5.9 void I2C_SlaveTransferCreateHandle (I2C_Type ∗ *base*, i2c_slave_handle_t ∗ *handle*, i2c_slave_transfer_callback_t *callback*, void ∗ *userData*)

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I2C_SlaveTransferAbort() API shall be called.

Parameters

	base	The I2C peripheral base address.
out	*handle*	Pointer to the I2C slave driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

19.5.5.10 status_t I2C_SlaveTransferNonBlocking (I2C_Type * *base*, i2c_slave_handle_t * *handle*, uint32_t *eventMask*)

Call this API after calling I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes kI2C_SlaveTransmitEvent callback. If no slave Rx transfer is busy, a master write to slave request invokes kI2C_SlaveReceiveEvent callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
handle	Pointer to i2c_slave_handle_t structure which stores the transfer state.
eventMask	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

19.5.5.11 status_t I2C_SlaveSetSendBuffer (I2C_Type * *base*, volatile i2c_slave_transfer_t * *transfer*, const void * *txData*, size_t *txSize*, uint32_t *eventMask*)

The function can be called in response to kI2C_SlaveTransmitEvent callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
transfer	Pointer to i2c_slave_transfer_t structure.
txData	Pointer to data to send to master.
txSize	Size of txData in bytes.
eventMask	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

19.5.5.12 status_t I2C_SlaveSetReceiveBuffer (I2C_Type * *base*, volatile i2c_slave_transfer_t * *transfer*, void * *rxData*, size_t *rxSize*, uint32_t *eventMask*)

The function can be called in response to kI2C_SlaveReceiveEvent callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
transfer	Pointer to i2c_slave_transfer_t structure.
rxData	Pointer to data to store data from master.
rxSize	Size of rxData in bytes.
eventMask	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

19.5.5.13 static uint32_t I2C_SlaveGetReceivedAddress (I2C_Type ∗ *base,* volatile i2c_slave_transfer_t ∗ *transfer*) [inline], [static]

This function should only be called from the address match event callback kI2C_SlaveAddressMatchEvent.

Parameters

base	The I2C peripheral base address.
transfer	The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

19.5.5.14 void I2C_SlaveTransferAbort (I2C_Type ∗ *base,* i2c_slave_handle_t ∗ *handle*)

Note

This API could be called at any time to stop slave for handling the bus events.

Parameters

base	The I2C peripheral base address.
handle	Pointer to i2c_slave_handle_t structure which stores the transfer state.

Return values

kStatus_Success	
kStatus_I2C_Idle	

19.5.5.15 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.

19.5.5.16 void I2C_SlaveTransferHandleIRQ (I2C_Type ∗ *base*, void ∗ *i2cHandle*)

Note

This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

base	The I2C peripheral base address.
handle	Pointer to i2c_slave_handle_t structure which stores the transfer state.

19.5.6 Variable Documentation

19.5.6.1 i2c_isr_t s_i2cMasterIsr

19.5.6.2 void∗ s_i2cHandle[FSL_FEATURE_SOC_I2C_COUNT]

19.6 I2C DMA Driver

19.6.1 Overview

Data Structures

- struct i2c_master_dma_handle_t
  I2C master dma transfer structure. More...

Macros

- #define I2C_MAX_DMA_TRANSFER_COUNT 1024
  Maximum lenght of single DMA transfer (determined by capability of the DMA engine)

Typedefs

- typedef void(∗ i2c_master_dma_transfer_callback_t)(I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, status_t status, void ∗userData)
  I2C master dma transfer callback typedef.

Driver version

- #define FSL_I2C_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  I2C DMA driver version 2.0.1.

I2C Block DMA Transfer Operation

- void I2C_MasterTransferCreateHandleDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, i2c_master_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Init the I2C handle which is used in transactional functions.
- status_t I2C_MasterTransferDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, i2c_-master_transfer_t ∗xfer)
  Performs a master dma non-blocking transfer on the I2C bus.
- status_t I2C_MasterTransferGetCountDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, size_t ∗count)
  Get master transfer status during a dma non-blocking transfer.
- void I2C_MasterTransferAbortDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle)
  Abort a master dma non-blocking transfer in a early time.

19.6.2 Data Structure Documentation

19.6.2.1 struct _i2c_master_dma_handle

I2C master dma handle typedef.

Data Fields

- uint8_t state
  Transfer state machine current state.
- uint32_t transferCount
  Indicates progress of the transfer.
- uint32_t remainingBytesDMA
  Remaining byte count to be transferred using DMA.
- uint8_t ∗ buf
  Buffer pointer for current state.
- dma_handle_t ∗ dmaHandle
  The DMA handler used.
- i2c_master_transfer_t transfer
  Copy of the current transfer info.
- i2c_master_dma_transfer_callback_t completionCallback
  Callback function called after dma transfer finished.
- void ∗ userData
  Callback parameter passed to callback function.

19.6.2.1.0.19 Field Documentation

19.6.2.1.0.19.1 uint8_t i2c_master_dma_handle_t::state

19.6.2.1.0.19.2 uint32_t i2c_master_dma_handle_t::remainingBytesDMA

19.6.2.1.0.19.3 uint8_t* i2c_master_dma_handle_t::buf

19.6.2.1.0.19.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

19.6.2.1.0.19.5 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

19.6.2.1.0.19.6 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

19.6.2.1.0.19.7 void* i2c_master_dma_handle_t::userData

19.6.3 Macro Definition Documentation

19.6.3.1 #define FSL_I2C_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

19.6.4 Typedef Documentation

19.6.4.1 typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)

19.6.5 Function Documentation

19.6.5.1 void I2C_MasterTransferCreateHandleDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*, i2c_master_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

Parameters

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

19.6.5.2 status_t I2C_MasterTransferDMA (I2C_Type ∗ *base*, i2c_master_dma_handle_t ∗ *handle*, i2c_master_transfer_t ∗ *xfer*)

Parameters

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

Return values

kStatus_Success	Sucessully complete the data transmission.
kStatus_I2C_Busy	Previous transmission still not finished.
kStatus_I2C_Timeout	Transfer error, wait signal timeout.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive Nak during transfer.

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

Parameters

base	I2C peripheral base address
handle	pointer to i2c_master_dma_handle_t structure

count	Number of bytes transferred so far by the non-blocking transaction.

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

Parameters

base	I2C peripheral base address
handle	pointer to i2c_master_dma_handle_t structure

19.7 I2C FreeRTOS Driver

19.7.1 Overview

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

Driver version

- #define FSL_I2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  I2C freertos driver version 2.0.1.

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t ∗handle, I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t ∗handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t ∗handle, i2c_master_transfer_t ∗transfer)
  Performs I2C transfer.

19.7.2 Data Structure Documentation

19.7.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type ∗ base
  I2C base address.
- i2c_master_handle_t drv_handle
  A handle of the underlying driver, treated as opaque by the RTOS layer.
- status_t async_status
  Transactional state of the underlying driver.
- SemaphoreHandle_t mutex
  A mutex to lock the handle during a transfer.
- SemaphoreHandle_t semaphore
  A semaphore to notify and unblock task when the transfer ends.

19.7.3 Macro Definition Documentation

19.7.3.1 #define FSL_I2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

19.7.4 Function Documentation

19.7.4.1 status_t I2C_RTOS_Init (i2c_rtos_handle_t * *handle*, I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	Configuration structure to set-up I2C in master mode.
srcClock_Hz	Frequency of input clock of the I2C module.

Returns

status of the operation.

19.7.4.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle.

19.7.4.3 status_t I2C_RTOS_Transfer (i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)

This function performs an I2C transfer according to data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 20
MRT: Multi-Rate Timer

20.1 Overview

The MCUXpresso SDK provides a driver for the Multi-Rate Timer (MRT) of MCUXpresso SDK devices.

20.2 Function groups

The MRT driver supports operating the module as a time counter.

20.2.1 Initialization and deinitialization

The function MRT_Init() initializes the MRT with specified configurations. The function MRT_GetDefaultConfig() gets the default configurations. The initialization function configures the MRT operating mode.

The function MRT_Deinit() stops the MRT timers and disables the module clock.

20.2.2 Timer period Operations

The function MRT_UpdateTimerPeriod() is used to update the timer period in units of count. The new value is immediately loaded or will be loaded at the end of the current time interval.

The function MRT_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. The user can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds

20.2.3 Start and Stop timer operations

The function MRT_StartTimer() starts the timer counting. After calling this function, the timer loads the period value, counts down to 0 and depending on the timer mode it either loads the respective start value again or stop. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function MRT_StopTimer() stops the timer counting.

20.2.4 Get and release channel

These functions can be used to reserve and release a channel. The function MRT_GetIdleChannel() finds the available channel. This function returns the lowest available channel number. The function MRT_-ReleaseChannel() release the channel when the timer is using the multi-task mode. In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use.

20.2.5 Status

Provides functions to get and clear the PIT status.

20.2.6 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

20.3 Typical use case

20.3.1 MRT tick example

Updates the MRT period and toggles an LED periodically. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/mrt

Files

- file fsl_mrt.h

Data Structures

- struct mrt_config_t
  MRT configuration structure. More...

Enumerations

- enum mrt_chnl_t {
  kMRT_Channel_0 = 0U,
  kMRT_Channel_1,
  kMRT_Channel_2,
  kMRT_Channel_3 }
  List of MRT channels.
- enum mrt_timer_mode_t {
  kMRT_RepeatMode = (0 << MRT_CHANNEL_CTRL_MODE_SHIFT),
  kMRT_OneShotMode = (1 << MRT_CHANNEL_CTRL_MODE_SHIFT),
  kMRT_OneShotStallMode = (2 << MRT_CHANNEL_CTRL_MODE_SHIFT) }
  List of MRT timer modes.

- enum mrt_interrupt_enable_t { kMRT_TimerInterruptEnable = MRT_CHANNEL_CTRL_INTEN_MASK }
  List of MRT interrupts.
- enum mrt_status_flags_t {
  kMRT_TimerInterruptFlag = MRT_CHANNEL_STAT_INTFLAG_MASK,
  kMRT_TimerRunFlag = MRT_CHANNEL_STAT_RUN_MASK }
  List of MRT status flags.

Driver version

- #define FSL_MRT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  Version 2.0.1.

Initialization and deinitialization

- void MRT_Init (MRT_Type *base, const mrt_config_t *config)
  Ungates the MRT clock and configures the peripheral for basic operation.
- void MRT_Deinit (MRT_Type *base)
  Gate the MRT clock.
- static void MRT_GetDefaultConfig (mrt_config_t *config)
  Fill in the MRT config struct with the default settings.
- static void MRT_SetupChannelMode (MRT_Type *base, mrt_chnl_t channel, const mrt_timer_mode_t mode)
  Sets up an MRT channel mode.

Interrupt Interface

- static void MRT_EnableInterrupts (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Enables the MRT interrupt.
- static void MRT_DisableInterrupts (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Disables the selected MRT interrupt.
- static uint32_t MRT_GetEnabledInterrupts (MRT_Type *base, mrt_chnl_t channel)
  Gets the enabled MRT interrupts.

Status Interface

- static uint32_t MRT_GetStatusFlags (MRT_Type *base, mrt_chnl_t channel)
  Gets the MRT status flags.
- static void MRT_ClearStatusFlags (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Clears the MRT status flags.

Read and Write the timer period

- void MRT_UpdateTimerPeriod (MRT_Type *base, mrt_chnl_t channel, uint32_t count, bool immediateLoad)
  Used to update the timer period in units of count.
- static uint32_t MRT_GetCurrentTimerCount (MRT_Type *base, mrt_chnl_t channel)
  Reads the current timer counting value.

Timer Start and Stop

- static void MRT_StartTimer (MRT_Type ∗base, mrt_chnl_t channel, uint32_t count)
  Starts the timer counting.
- static void MRT_StopTimer (MRT_Type ∗base, mrt_chnl_t channel)
  Stops the timer counting.

Get & release channel

- static uint32_t MRT_GetIdleChannel (MRT_Type ∗base)
  Find the available channel.

20.4 Data Structure Documentation

20.4.1 struct mrt_config_t

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

The config struct can be made const so it resides in flash

Data Fields

- bool enableMultiTask
  true: Timers run in multi-task mode; false: Timers run in hardware status mode

20.5 Enumeration Type Documentation

20.5.1 enum mrt_chnl_t

Enumerator

kMRT_Channel_0 MRT channel number 0.
kMRT_Channel_1 MRT channel number 1.
kMRT_Channel_2 MRT channel number 2.
kMRT_Channel_3 MRT channel number 3.

20.5.2 enum mrt_timer_mode_t

Enumerator

kMRT_RepeatMode Repeat Interrupt mode.
kMRT_OneShotMode One-shot Interrupt mode.
kMRT_OneShotStallMode One-shot stall mode.

20.5.3 enum mrt_interrupt_enable_t

Enumerator

kMRT_TimerInterruptEnable Timer interrupt enable.

20.5.4 enum mrt_status_flags_t

Enumerator

kMRT_TimerInterruptFlag Timer interrupt flag.
kMRT_TimerRunFlag Indicates state of the timer.

20.6 Function Documentation

20.6.1 void MRT_Init (MRT_Type ∗ *base*, const mrt_config_t ∗ *config*)

Note

This API should be called at the beginning of the application using the MRT driver.

Parameters

base	Multi-Rate timer peripheral base address
config	Pointer to user's MRT config structure. If MRT has MULTITASK bit field in MOD-CFG reigster, param config is useless.

20.6.2 void MRT_Deinit (MRT_Type ∗ *base*)

Parameters

base	Multi-Rate timer peripheral base address

20.6.3 static void MRT_GetDefaultConfig (mrt_config_t ∗ *config*) [inline], [static]

The default values are:

```
*     config->enableMultiTask = false;
*
```

Parameters

config	Pointer to user's MRT config structure.

20.6.4 static void MRT_SetupChannelMode (MRT_Type * *base*, mrt_chnl_t *channel*, const mrt_timer_mode_t *mode*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Channel that is being configured.
mode	Timer mode to use for the channel.

20.6.5 static void MRT_EnableInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The interrupts to enable. This is a logical OR of members of the enumeration mrt_interrupt_enable_t

20.6.6 static void MRT_DisableInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The interrupts to disable. This is a logical OR of members of the enumeration mrt_interrupt_enable_t

20.6.7 static uint32_t MRT_GetEnabledInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

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

20.6.8 static uint32_t MRT_GetStatusFlags (MRT_Type ∗ *base*, mrt_chnl_t *channel*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

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

20.6.9 static void MRT_ClearStatusFlags (MRT_Type ∗ *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The status flags to clear. This is a logical OR of members of the enumeration mrt_status_flags_t

20.6.10 void MRT_UpdateTimerPeriod (MRT_Type ∗ *base*, mrt_chnl_t *channel*, uint32_t *count*, bool *immediateLoad*)

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
count	Timer period in units of ticks
immediateLoad	true: Load the new value immediately into the TIMER register; false: Load the new value at the end of current timer interval

20.6.11 static uint32_t MRT_GetCurrentTimerCount (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

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

Note

User can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

Current timer counting value in ticks

20.6.12 static void MRT_StartTimer (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *count*) [inline], [static]

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.
count	Timer period in units of ticks

20.6.13 static void MRT_StopTimer (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

This function stops the timer from counting.

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.

20.6.14 static uint32_t MRT_GetIdleChannel (MRT_Type * *base*) [inline], [static]

This function returns the lowest available channel number.

Parameters

base	Multi-Rate timer peripheral base address

Chapter 21
INPUTMUX: Input Multiplexing Driver

21.1 Overview

The MCUXpresso SDK provides a driver for the Input multiplexing (INPUTMUX).

It configures the inputs to the pin interrupt block, DMA trigger, and frequency measure function. Once configured, the clock is not needed for the inputmux.

21.2 Input Multiplexing Driver operation

INPUTMUX_AttachSignal function configures the specified input

21.3 Typical use case

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

Files

- file fsl_inputmux.h
- file fsl_inputmux_connections.h

Functions

- void INPUTMUX_Init (INPUTMUX_Type *base)
  Initialize INPUTMUX peripheral.
- void INPUTMUX_AttachSignal (INPUTMUX_Type *base, uint32_t index, inputmux_connection-_t connection)
  Attaches a signal.
- void INPUTMUX_Deinit (INPUTMUX_Type *base)
  Deinitialize INPUTMUX peripheral.

Driver version

- #define FSL_INPUTMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Group interrupt driver version for SDK.

Input multiplexing connections

- enum inputmux_connection_t {
  kINPUTMUX_DmaChannel0TrigoutToTriginChannels = 0U + (DMA_OTRIG_PMUX_ID << P-MUX_SHIFT) ,
  kINPUTMUX_DmaChannel24TrigoutToTriginChannels = 24U + (DMA_OTRIG_PMUX_ID << PMUX_SHIFT) ,
  kINPUTMUX_DebugHaltedToSct0 = 9U + (SCT0_PMUX_ID << PMUX_SHIFT) }

INPUTMUX connections type.
- #define DMA_OTRIG_PMUX_ID 0x00U
  Periphinmux IDs.
- #define **SCT0_PMUX_ID** 0x20U
- #define **DMA_TRIG0_PMUX_ID** 0x40U
- #define **PMUX_SHIFT** 20U

21.4 Macro Definition Documentation

21.4.1 #define FSL_INPUTMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

21.5 Enumeration Type Documentation

21.5.1 enum inputmux_connection_t

Enumerator

kINPUTMUX_DmaChannel0TrigoutToTriginChannels DMA OTRIG.
kINPUTMUX_DmaChannel24TrigoutToTriginChannels SCT INMUX.
kINPUTMUX_DebugHaltedToSct0 DMA ITRIG.

21.6 Function Documentation

21.6.1 void INPUTMUX_Init (INPUTMUX_Type ∗ *base*)

This function enables the INPUTMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.

Return values

None.	

21.6.2 void INPUTMUX_AttachSignal (INPUTMUX_Type ∗ *base,* uint32_t *index,* inputmux_connection_t *connection*)

This function gates the INPUTPMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.
index	Destination peripheral to attach the signal to.
connection	Selects connection.

Return values

None.	

21.6.3 void INPUTMUX_Deinit (INPUTMUX_Type * *base*)

This function disables the INPUTMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.

Return values

None.	

Chapter 22
SWM: Switch Matrix Module

22.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Switch Matrix Module (SWM) module of MCUXpresso SDK devices.

The function SWM_SetMovablePinSelect() will selects a movable pin designated by its GPIO port and bit numbers to a function.

The function SWM_SetFixedMovablePinSelect() will selects a fixed movable pin designated by its GPIO port and bit numbers to a function.

The function SWM_SetFixedPinSelect() will enables a fixed-pin function in PINENABLE0 or PINENABLE1.

Files

- file fsl_swm.h

Functions

- void SWM_SetMovablePinSelect (SWM_Type *base, swm_select_movable_t func, swm_port_pin_type_t swm_port_pin)
  Assignment of digital peripheral functions to pins.
- void SWM_SetFixedPinSelect (SWM_Type *base, swm_select_fixed_pin_t func, bool enable)
  Enable the fixed-pin function.

Driver version

- #define FSL_SWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

swm connections

- enum swm_port_pin_type_t {
  kSWM_PortPin_P0_0 = 0U,
  kSWM_PortPin_P0_1 = 1U,
  kSWM_PortPin_P0_2 = 2U,
  kSWM_PortPin_P0_3 = 3U,
  kSWM_PortPin_P0_4 = 4U,
  kSWM_PortPin_P0_5 = 5U,
  kSWM_PortPin_P0_6 = 6U,
  kSWM_PortPin_P0_7 = 7U,
  kSWM_PortPin_P0_8 = 8U,
  kSWM_PortPin_P0_9 = 9U,
  kSWM_PortPin_P0_10 = 10U,
  kSWM_PortPin_P0_11 = 11U,
  kSWM_PortPin_P0_12 = 12U,
  kSWM_PortPin_P0_13 = 13U,
  kSWM_PortPin_P0_14 = 14U,
  kSWM_PortPin_P0_15 = 15U,
  kSWM_PortPin_P0_16 = 16U,
  kSWM_PortPin_P0_17 = 17U,
  kSWM_PortPin_P0_18 = 18U,
  kSWM_PortPin_P0_19 = 19U,
  kSWM_PortPin_P0_20 = 20U,
  kSWM_PortPin_P0_21 = 21U,
  kSWM_PortPin_P0_22 = 22U,
  kSWM_PortPin_P0_23 = 23U,
  kSWM_PortPin_P0_24 = 24U,
  kSWM_PortPin_P0_25 = 25U,
  kSWM_PortPin_P0_26 = 26U,
  kSWM_PortPin_P0_27 = 27U,
  kSWM_PortPin_P0_28 = 28U,
  kSWM_PortPin_P0_29 = 29U,
  kSWM_PortPin_P0_30 = 30U,
  kSWM_PortPin_P0_31 = 31U,
  kSWM_PortPin_P1_0 = 32U,
  kSWM_PortPin_P1_1 = 33U,
  kSWM_PortPin_P1_2 = 34U,
  kSWM_PortPin_P1_3 = 35U,
  kSWM_PortPin_P1_4 = 36U,
  kSWM_PortPin_P1_5 = 37U,
  kSWM_PortPin_P1_6 = 38U,
  kSWM_PortPin_P1_7 = 39U,
  kSWM_PortPin_P1_8 = 40U,
  kSWM_PortPin_P1_9 = 41U,
  kSWM_PortPin_P1_10 = 42U,
  kSWM_PortPin_P1_11 = 43U,
  kSWM_PortPin_P1_12 = 44U,
  kSWM_PortPin_P1_13 = 45U,
  kSWM_PortPin_P1_14 = 46U,

kSWM_PortPin_P1_21 = 53U }
SWM port_pin number.
- enum swm_select_movable_t {

kSWM_USART0_TXD = 0U,
kSWM_USART0_RXD = 1U,
kSWM_USART0_RTS = 2U,
kSWM_USART0_CTS = 3U,
kSWM_USART0_SCLK = 4U,
kSWM_USART1_TXD = 5U,
kSWM_USART1_RXD = 6U,
kSWM_USART1_RTS = 7U,
kSWM_USART1_CTS = 8U,
kSWM_USART1_SCLK = 9U,
kSWM_USART2_TXD = 10U,
kSWM_USART2_RXD = 11U,
kSWM_USART2_RTS = 12U,
kSWM_USART2_CTS = 13U,
kSWM_USART2_SCLK = 14U,
kSWM_SPI0_SCK = 15U,
kSWM_SPI0_MOSI = 16U,
kSWM_SPI0_MISO = 17U,
kSWM_SPI0_SSEL0 = 18U,
kSWM_SPI0_SSEL1 = 19U,
kSWM_SPI0_SSEL2 = 20U,
kSWM_SPI0_SSEL3 = 21U,
kSWM_SPI1_SCK = 22U,
kSWM_SPI1_MOSI = 23U,
kSWM_SPI1_MISO = 24U,
kSWM_SPI1_SSEL0 = 25U,
kSWM_SPI1_SSEL1 = 26U,
kSWM_SCT_PIN0 = 27U,
kSWM_SCT_PIN1 = 28U,
kSWM_SCT_PIN2 = 29U,
kSWM_SCT_PIN3 = 30U,
kSWM_SCT_OUT0 = 31U,
kSWM_SCT_OUT1 = 32U,
kSWM_SCT_OUT2 = 33U,
kSWM_SCT_OUT3 = 34U,
kSWM_SCT_OUT4 = 35U,
kSWM_SCT_OUT5 = 36U,
kSWM_SCT_OUT6 = 37U,
kSWM_I2C1_SDA = 38U,
kSWM_I2C1_SCL = 39U,
kSWM_I2C2_SDA = 40U,
kSWM_I2C2_SCL = 41U,
kSWM_I2C3_SDA = 42U,
kSWM_I2C3_SCL = 43U,
kSWM_ACMP_OUT = 44U,
kSWM_CLKOUT = 45U,
kSWM_GPIO_INT_BMAT = 46U,
kSWM_USART3_TXD = 47U,
kSWM_USART3_RXD = 48U,

kSWM_MOVABLE_NUM_FUNCS = 60U }
SWM movable selection.
- enum swm_select_fixed_pin_t {
kSWM_ACMP_INPUT1 = SWM_PINENABLE0_ACMP_I1_MASK,
kSWM_ACMP_INPUT2 = SWM_PINENABLE0_ACMP_I2_MASK,
kSWM_ACMP_INPUT3 = SWM_PINENABLE0_ACMP_I3_MASK,
kSWM_ACMP_INPUT4 = SWM_PINENABLE0_ACMP_I4_MASK,
kSWM_ACMP_INPUT5 = SWM_PINENABLE0_ACMP_I5_MASK,
kSWM_SWCLK = SWM_PINENABLE0_SWCLK_MASK,
kSWM_SWDIO = SWM_PINENABLE0_SWDIO_MASK,
kSWM_XTALIN = SWM_PINENABLE0_XTALIN_MASK,
kSWM_XTALOUT = SWM_PINENABLE0_XTALOUT_MASK,
kSWM_RESETN = SWM_PINENABLE0_RESETN_MASK,
kSWM_CLKIN = SWM_PINENABLE0_CLKIN_MASK,
kSWM_VDDCMP = SWM_PINENABLE0_VDDCMP_MASK,
kSWM_I2C0_SDA = SWM_PINENABLE0_I2C0_SDA_MASK,
kSWM_I2C0_SCL = SWM_PINENABLE0_I2C0_SCL_MASK,
kSWM_ADC_CHN0 = SWM_PINENABLE0_ADC_0_MASK,
kSWM_ADC_CHN1 = SWM_PINENABLE0_ADC_1_MASK,
kSWM_ADC_CHN2 = SWM_PINENABLE0_ADC_2_MASK,
kSWM_ADC_CHN3 = SWM_PINENABLE0_ADC_3_MASK,
kSWM_ADC_CHN4 = SWM_PINENABLE0_ADC_4_MASK,
kSWM_ADC_CHN5 = SWM_PINENABLE0_ADC_5_MASK,
kSWM_ADC_CHN6 = SWM_PINENABLE0_ADC_6_MASK,
kSWM_ADC_CHN7 = SWM_PINENABLE0_ADC_7_MASK,
kSWM_ADC_CHN8 = SWM_PINENABLE0_ADC_8_MASK,
kSWM_ADC_CHN9 = SWM_PINENABLE0_ADC_9_MASK,
kSWM_ADC_CHN10 = SWM_PINENABLE0_ADC_10_MASK,
kSWM_ADC_CHN11 = SWM_PINENABLE0_ADC_11_MASK,
kSWM_DAC_OUT0 = SWM_PINENABLE0_DACOUT0_MASK,
kSWM_DAC_OUT1 = SWM_PINENABLE0_DACOUT1_MASK,
kSWM_CAPT_X0,
kSWM_CAPT_X1 = SWM_PINENABLE0_CAPT_X1_MASK,
kSWM_CAPT_X2 = SWM_PINENABLE0_CAPT_X2_MASK,
kSWM_CAPT_X3 = (int)SWM_PINENABLE0_CAPT_X3_MASK,
kSWM_CAPT_X4 = (int)(SWM_PINENABLE1_CAPT_X4_MASK | 0x80000000),
kSWM_CAPT_X5 = (int)(SWM_PINENABLE1_CAPT_X5_MASK | 0x80000000),
kSWM_CAPT_X6 = (int)(SWM_PINENABLE1_CAPT_X6_MASK | 0x80000000),
kSWM_CAPT_X7 = (int)(SWM_PINENABLE1_CAPT_X7_MASK | 0x80000000),
kSWM_CAPT_X8 = (int)(SWM_PINENABLE1_CAPT_X8_MASK | 0x80000000),
kSWM_CAPT_YL,
kSWM_CAPT_YH = (int)(SWM_PINENABLE1_CAPT_YH_MASK | 0x80000000),
kSWM_FIXEDPIN_NUM_FUNCS = (int)0x80000041 }
SWM fixed pin selection.

22.2 Enumeration Type Documentation

22.2.1 enum swm_port_pin_type_t

Enumerator

kSWM_PortPin_P0_0 port_pin number P0_0.
kSWM_PortPin_P0_1 port_pin number P0_1.
kSWM_PortPin_P0_2 port_pin number P0_2.
kSWM_PortPin_P0_3 port_pin number P0_3.
kSWM_PortPin_P0_4 port_pin number P0_4.
kSWM_PortPin_P0_5 port_pin number P0_5.
kSWM_PortPin_P0_6 port_pin number P0_6.
kSWM_PortPin_P0_7 port_pin number P0_7.
kSWM_PortPin_P0_8 port_pin number P0_8.
kSWM_PortPin_P0_9 port_pin number P0_9.
kSWM_PortPin_P0_10 port_pin number P0_10.
kSWM_PortPin_P0_11 port_pin number P0_11.
kSWM_PortPin_P0_12 port_pin number P0_12.
kSWM_PortPin_P0_13 port_pin number P0_13.
kSWM_PortPin_P0_14 port_pin number P0_14.
kSWM_PortPin_P0_15 port_pin number P0_15.
kSWM_PortPin_P0_16 port_pin number P0_16.
kSWM_PortPin_P0_17 port_pin number P0_17.
kSWM_PortPin_P0_18 port_pin number P0_18.
kSWM_PortPin_P0_19 port_pin number P0_19.
kSWM_PortPin_P0_20 port_pin number P0_20.
kSWM_PortPin_P0_21 port_pin number P0_21.
kSWM_PortPin_P0_22 port_pin number P0_22.
kSWM_PortPin_P0_23 port_pin number P0_23.
kSWM_PortPin_P0_24 port_pin number P0_24.
kSWM_PortPin_P0_25 port_pin number P0_25.
kSWM_PortPin_P0_26 port_pin number P0_26.
kSWM_PortPin_P0_27 port_pin number P0_27.
kSWM_PortPin_P0_28 port_pin number P0_28.
kSWM_PortPin_P0_29 port_pin number P0_29.
kSWM_PortPin_P0_30 port_pin number P0_30.
kSWM_PortPin_P0_31 port_pin number P0_31.
kSWM_PortPin_P1_0 port_pin number P1_0.
kSWM_PortPin_P1_1 port_pin number P1_1.
kSWM_PortPin_P1_2 port_pin number P1_2.
kSWM_PortPin_P1_3 port_pin number P1_3.
kSWM_PortPin_P1_4 port_pin number P1_4.
kSWM_PortPin_P1_5 port_pin number P1_5.
kSWM_PortPin_P1_6 port_pin number P1_6.
kSWM_PortPin_P1_7 port_pin number P1_7.

kSWM_PortPin_P1_8 port_pin number P1_8.
kSWM_PortPin_P1_9 port_pin number P1_9.
kSWM_PortPin_P1_10 port_pin number P1_10.
kSWM_PortPin_P1_11 port_pin number P1_11.
kSWM_PortPin_P1_12 port_pin number P1_12.
kSWM_PortPin_P1_13 port_pin number P1_13.
kSWM_PortPin_P1_14 port_pin number P1_14.
kSWM_PortPin_P1_15 port_pin number P1_15.
kSWM_PortPin_P1_16 port_pin number P1_16.
kSWM_PortPin_P1_17 port_pin number P1_17.
kSWM_PortPin_P1_18 port_pin number P1_18.
kSWM_PortPin_P1_19 port_pin number P1_19.
kSWM_PortPin_P1_20 port_pin number P1_20.
kSWM_PortPin_P1_21 port_pin number P1_21.

22.2.2 enum swm_select_movable_t

Enumerator

kSWM_USART0_TXD Movable function as USART0_TXD.
kSWM_USART0_RXD Movable function as USART0_RXD.
kSWM_USART0_RTS Movable function as USART0_RTS.
kSWM_USART0_CTS Movable function as USART0_CTS.
kSWM_USART0_SCLK Movable function as USART0_SCLK.
kSWM_USART1_TXD Movable function as USART1_TXD.
kSWM_USART1_RXD Movable function as USART1_RXD.
kSWM_USART1_RTS Movable function as USART1_RTS.
kSWM_USART1_CTS Movable function as USART1_CTS.
kSWM_USART1_SCLK Movable function as USART1_SCLK.
kSWM_USART2_TXD Movable function as USART2_TXD.
kSWM_USART2_RXD Movable function as USART2_RXD.
kSWM_USART2_RTS Movable function as USART2_RTS.
kSWM_USART2_CTS Movable function as USART2_CTS.
kSWM_USART2_SCLK Movable function as USART2_SCLK.
kSWM_SPI0_SCK Movable function as SPI0_SCK.
kSWM_SPI0_MOSI Movable function as SPI0_MOSI.
kSWM_SPI0_MISO Movable function as SPI0_MISO.
kSWM_SPI0_SSEL0 Movable function as SPI0_SSEL0.
kSWM_SPI0_SSEL1 Movable function as SPI0_SSEL1.
kSWM_SPI0_SSEL2 Movable function as SPI0_SSEL2.
kSWM_SPI0_SSEL3 Movable function as SPI0_SSEL3.
kSWM_SPI1_SCK Movable function as SPI1_SCK.
kSWM_SPI1_MOSI Movable function as SPI1_MOSI.
kSWM_SPI1_MISO Movable function as SPI1_MISO.

kSWM_SPI1_SSEL0 Movable function as SPI1_SSEL0.
kSWM_SPI1_SSEL1 Movable function as SPI1_SSEL1.
kSWM_SCT_PIN0 Movable function as SCT_PIN0.
kSWM_SCT_PIN1 Movable function as SCT_PIN1.
kSWM_SCT_PIN2 Movable function as SCT_PIN2.
kSWM_SCT_PIN3 Movable function as SCT_PIN3.
kSWM_SCT_OUT0 Movable function as SCT_OUT0.
kSWM_SCT_OUT1 Movable function as SCT_OUT1.
kSWM_SCT_OUT2 Movable function as SCT_OUT2.
kSWM_SCT_OUT3 Movable function as SCT_OUT3.
kSWM_SCT_OUT4 Movable function as SCT_OUT4.
kSWM_SCT_OUT5 Movable function as SCT_OUT5.
kSWM_SCT_OUT6 Movable function as SCT_OUT6.
kSWM_I2C1_SDA Movable function as I2C1_SDA.
kSWM_I2C1_SCL Movable function as I2C1_SCL.
kSWM_I2C2_SDA Movable function as I2C2_SDA.
kSWM_I2C2_SCL Movable function as I2C2_SCL.
kSWM_I2C3_SDA Movable function as I2C3_SDA.
kSWM_I2C3_SCL Movable function as I2C3_SCL.
kSWM_ACMP_OUT Movable function as ACMP_OUT.
kSWM_CLKOUT Movable function as CLKOUT.
kSWM_GPIO_INT_BMAT Movable function as GPIO_INT_BMAT.
kSWM_USART3_TXD Movable function as USART3_TXD.
kSWM_USART3_RXD Movable function as USART3_RXD.
kSWM_USART3_SCLK Movable function as USART3_SCLK.
kSWM_USART4_TXD Movable function as USART4_TXD.
kSWM_USART4_RXD Movable function as USART4_RXD.
kSWM_USART4_SCLK Movable function as USART4_SCLK.
kSWM_T0_MAT_CHN0 Movable function as Timer Match Channel 0.
kSWM_T0_MAT_CHN1 Movable function as Timer Match Channel 1.
kSWM_T0_MAT_CHN2 Movable function as Timer Match Channel 2.
kSWM_T0_MAT_CHN3 Movable function as Timer Match Channel 3.
kSWM_T0_CAP_CHN0 Movable function as Timer Capture Channel 0.
kSWM_T0_CAP_CHN1 Movable function as Timer Capture Channel 1.
kSWM_T0_CAP_CHN2 Movable function as Timer Capture Channel 2.
kSWM_MOVABLE_NUM_FUNCS Movable function number.

22.2.3 enum swm_select_fixed_pin_t

Enumerator

kSWM_ACMP_INPUT1 Fixed-pin function as ACMP_INPUT1.
kSWM_ACMP_INPUT2 Fixed-pin function as ACMP_INPUT2.
kSWM_ACMP_INPUT3 Fixed-pin function as ACMP_INPUT3.

kSWM_ACMP_INPUT4 Fixed-pin function as ACMP_INPUT4.
kSWM_ACMP_INPUT5 Fixed-pin function as ACMP_INPUT5.
kSWM_SWCLK Fixed-pin function as SWCLK.
kSWM_SWDIO Fixed-pin function as SWDIO.
kSWM_XTALIN Fixed-pin function as XTALIN.
kSWM_XTALOUT Fixed-pin function as XTALOUT.
kSWM_RESETN Fixed-pin function as RESETN.
kSWM_CLKIN Fixed-pin function as CLKIN.
kSWM_VDDCMP Fixed-pin function as VDDCMP.
kSWM_I2C0_SDA Fixed-pin function as I2C0_SDA.
kSWM_I2C0_SCL Fixed-pin function as I2C0_SCL.
kSWM_ADC_CHN0 Fixed-pin function as ADC_CHN0.
kSWM_ADC_CHN1 Fixed-pin function as ADC_CHN1.
kSWM_ADC_CHN2 Fixed-pin function as ADC_CHN2.
kSWM_ADC_CHN3 Fixed-pin function as ADC_CHN3.
kSWM_ADC_CHN4 Fixed-pin function as ADC_CHN4.
kSWM_ADC_CHN5 Fixed-pin function as ADC_CHN5.
kSWM_ADC_CHN6 Fixed-pin function as ADC_CHN6.
kSWM_ADC_CHN7 Fixed-pin function as ADC_CHN7.
kSWM_ADC_CHN8 Fixed-pin function as ADC_CHN8.
kSWM_ADC_CHN9 Fixed-pin function as ADC_CHN9.
kSWM_ADC_CHN10 Fixed-pin function as ADC_CHN10.
kSWM_ADC_CHN11 Fixed-pin function as ADC_CHN11.
kSWM_DAC_OUT0 Fixed-pin function as DACOUT0.
kSWM_DAC_OUT1 Fixed-pin function as DACOUT1.
kSWM_CAPT_X0 Fixed-pin function as CAPT_X0, an X capacitor(a mutual capacitance touch sensor) .
kSWM_CAPT_X1 Fixed-pin function as CAPT_X1.
kSWM_CAPT_X2 Fixed-pin function as CAPT_X2.
kSWM_CAPT_X3 Fixed-pin function as CAPT_X3.
kSWM_CAPT_X4 Fixed-pin function as CAPT_X4.
kSWM_CAPT_X5 Fixed-pin function as CAPT_X5.
kSWM_CAPT_X6 Fixed-pin function as CAPT_X6.
kSWM_CAPT_X7 Fixed-pin function as CAPT_X7.
kSWM_CAPT_X8 Fixed-pin function as CAPT_X8.
kSWM_CAPT_YL Fixed-pin function as CAPT_YL, an Y capacitor(the measurement capacitor) .
kSWM_CAPT_YH Fixed-pin function as CAPT_YH.
kSWM_FIXEDPIN_NUM_FUNCS Fixed-pin function number.

22.3 Function Documentation

22.3.1 void SWM_SetMovablePinSelect (SWM_Type * *base*, swm_select_movable_t *func*, swm_port_pin_type_t *swm_port_pin*)

This function will selects a pin (designated by its GPIO port and bit numbers) to a function.

Parameters

base	SWM peripheral base address.
func	any function name that is movable.
swm_port_pin	any pin which has a GPIO port number and bit number.

22.3.2 void SWM_SetFixedPinSelect (SWM_Type * *base,* swm_select_fixed_pin_t *func,* bool *enable*)

This function will enables a fixed-pin function in PINENABLE0 or PINENABLE1.

Parameters

base	SWM peripheral base address.
func	any function name that is fixed pin.
enable	enable or disable.

Chapter 23
SCTimer: SCTimer/PWM (SCT)

23.1 Overview

The MCUXpresso SDK provides a driver for the SCTimer Module (SCT) of MCUXpresso SDK devices.

23.2 Function groups

The SCTimer driver supports the generation of PWM signals. The driver also supports enabling events in various states of the SCTimer and the actions that will be triggered when an event occurs.

23.2.1 Initialization and deinitialization

The function SCTIMER_Init() initializes the SCTimer with specified configurations. The function SCTIMER_GetDefaultConfig() gets the default configurations.

The function SCTIMER_Deinit() halts the SCTimer counter and turns off the module clock.

23.2.2 PWM Operations

The function SCTIMER_SetupPwm() sets up SCTimer channels for PWM output. The function can set up the PWM signal properties duty cycle and level-mode (active low or high) to use. However, the same PWM period and PWM mode (edge or center-aligned) 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 1 and 100.

The function SCTIMER_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular SCTimer channel.

23.2.3 Status

Provides functions to get and clear the SCTimer status.

23.2.4 Interrupt

Provides functions to enable/disable SCTimer interrupts and get current enabled interrupts.

23.3 SCTimer State machine and operations

The SCTimer has 10 states and each state can have a set of events enabled that can trigger a user specified action when the event occurs.

23.3.1 SCTimer event operations

The user can create an event and enable it in the current state using the functions SCTIMER_CreateAndScheduleEvent() and SCTIMER_ScheduleEvent(). SCTIMER_CreateAndScheduleEvent() creates a new event based on the users preference and enables it in the current state. SCTIMER_ScheduleEvent() enables an event created earlier in the current state.

23.3.2 SCTimer state operations

The user can get the current state number by calling SCTIMER_GetCurrentState(), he can use this state number to set state transitions when a particular event is triggered.

Once the user has created and enabled events for the current state he can go to the next state by calling the function SCTIMER_IncreaseState(). The user can then start creating events to be enabled in this new state.

23.3.3 SCTimer action operations

There are a set of functions that decide what action should be taken when an event is triggered. SCTIMER_SetupCaptureAction() sets up which counter to capture and which capture register to read on event trigger. SCTIMER_SetupNextStateAction() sets up which state the SCTimer state machine should transition to on event trigger. SCTIMER_SetupOutputSetAction() sets up which pin to set on event trigger. SCTIMER_SetupOutputClearAction() sets up which pin to clear on event trigger. SCTIMER_SetupOutputToggleAction() sets up which pin to toggle on event trigger. SCTIMER_SetupCounterLimitAction() sets up which counter will be limited on event trigger. SCTIMER_SetupCounterStopAction() sets up which counter will be stopped on event trigger. SCTIMER_SetupCounterStartAction() sets up which counter will be started on event trigger. SCTIMER_SetupCounterHaltAction() sets up which counter will be halted on event trigger. SCTIMER_SetupDmaTriggerAction() sets up which DMA request will be activated on event trigger.

23.4 16-bit counter mode

The SCTimer is configurable to run as two 16-bit counters via the enableCounterUnify flag that is available in the configuration structure passed in to the SCTIMER_Init() function.

When operating in 16-bit mode, it is important the user specify the appropriate counter to use when working with the functions: SCTIMER_StartTimer(), SCTIMER_StopTimer(), SCTIMER_CreateAndScheduleEvent(), SCTIMER_SetupCaptureAction(), SCTIMER_SetupCounterLimitAction(), SCTIM-

ER_SetupCounterStopAction(), SCTIMER_SetupCounterStartAction(), and SCTIMER_SetupCounterHaltAction().

23.5 Typical use case

23.5.1 PWM output

Output a PWM signal on 2 SCTimer channels with different duty cycles. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/sctimer

Files

- file fsl_sctimer.h

Data Structures

- struct sctimer_pwm_signal_param_t
  Options to configure a SCTimer PWM signal. More...
- struct sctimer_config_t
  SCTimer configuration structure. More...

Typedefs

- typedef void(∗ sctimer_event_callback_t)(void)
  SCTimer callback typedef.

Enumerations

- enum sctimer_pwm_mode_t {
  kSCTIMER_EdgeAlignedPwm = 0U,
  kSCTIMER_CenterAlignedPwm }
  SCTimer PWM operation modes.
- enum sctimer_counter_t {
  kSCTIMER_Counter_L = 0U,
  kSCTIMER_Counter_H }
  SCTimer counters when working as two independent 16-bit counters.
- enum sctimer_input_t {
  kSCTIMER_Input_0 = 0U,
  kSCTIMER_Input_1,
  kSCTIMER_Input_2,
  kSCTIMER_Input_3,
  kSCTIMER_Input_4,
  kSCTIMER_Input_5,
  kSCTIMER_Input_6,
  kSCTIMER_Input_7 }
  List of SCTimer input pins.

- enum sctimer_out_t {
  kSCTIMER_Out_0 = 0U,
  kSCTIMER_Out_1,
  kSCTIMER_Out_2,
  kSCTIMER_Out_3,
  kSCTIMER_Out_4,
  kSCTIMER_Out_5,
  kSCTIMER_Out_6,
  kSCTIMER_Out_7,
  kSCTIMER_Out_8,
  kSCTIMER_Out_9 }
  List of SCTimer output pins.
- enum sctimer_pwm_level_select_t {
  kSCTIMER_LowTrue = 0U,
  kSCTIMER_HighTrue }
  SCTimer PWM output pulse mode: high-true, low-true or no output.
- enum sctimer_clock_mode_t {
  kSCTIMER_System_ClockMode = 0U,
  kSCTIMER_Sampled_ClockMode,
  kSCTIMER_Input_ClockMode,
  kSCTIMER_Asynchronous_ClockMode }
  SCTimer clock mode options.
- enum sctimer_clock_select_t {
  kSCTIMER_Clock_On_Rise_Input_0 = 0U,
  kSCTIMER_Clock_On_Fall_Input_0,
  kSCTIMER_Clock_On_Rise_Input_1,
  kSCTIMER_Clock_On_Fall_Input_1,
  kSCTIMER_Clock_On_Rise_Input_2,
  kSCTIMER_Clock_On_Fall_Input_2,
  kSCTIMER_Clock_On_Rise_Input_3,
  kSCTIMER_Clock_On_Fall_Input_3,
  kSCTIMER_Clock_On_Rise_Input_4,
  kSCTIMER_Clock_On_Fall_Input_4,
  kSCTIMER_Clock_On_Rise_Input_5,
  kSCTIMER_Clock_On_Fall_Input_5,
  kSCTIMER_Clock_On_Rise_Input_6,
  kSCTIMER_Clock_On_Fall_Input_6,
  kSCTIMER_Clock_On_Rise_Input_7,
  kSCTIMER_Clock_On_Fall_Input_7 }
  SCTimer clock select options.
- enum sctimer_conflict_resolution_t {
  kSCTIMER_ResolveNone = 0U,
  kSCTIMER_ResolveSet,
  kSCTIMER_ResolveClear,
  kSCTIMER_ResolveToggle }
  SCTimer output conflict resolution options.

- enum sctimer_event_t
  List of SCTimer event types.
- enum sctimer_interrupt_enable_t {
  kSCTIMER_Event0InterruptEnable = (1U << 0),
  kSCTIMER_Event1InterruptEnable = (1U << 1),
  kSCTIMER_Event2InterruptEnable = (1U << 2),
  kSCTIMER_Event3InterruptEnable = (1U << 3),
  kSCTIMER_Event4InterruptEnable = (1U << 4),
  kSCTIMER_Event5InterruptEnable = (1U << 5),
  kSCTIMER_Event6InterruptEnable = (1U << 6),
  kSCTIMER_Event7InterruptEnable = (1U << 7),
  kSCTIMER_Event8InterruptEnable = (1U << 8),
  kSCTIMER_Event9InterruptEnable = (1U << 9),
  kSCTIMER_Event10InterruptEnable = (1U << 10),
  kSCTIMER_Event11InterruptEnable = (1U << 11),
  kSCTIMER_Event12InterruptEnable = (1U << 12) }
  List of SCTimer interrupts.
- enum sctimer_status_flags_t {
  kSCTIMER_Event0Flag = (1U << 0),
  kSCTIMER_Event1Flag = (1U << 1),
  kSCTIMER_Event2Flag = (1U << 2),
  kSCTIMER_Event3Flag = (1U << 3),
  kSCTIMER_Event4Flag = (1U << 4),
  kSCTIMER_Event5Flag = (1U << 5),
  kSCTIMER_Event6Flag = (1U << 6),
  kSCTIMER_Event7Flag = (1U << 7),
  kSCTIMER_Event8Flag = (1U << 8),
  kSCTIMER_Event9Flag = (1U << 9),
  kSCTIMER_Event10Flag = (1U << 10),
  kSCTIMER_Event11Flag = (1U << 11),
  kSCTIMER_Event12Flag = (1U << 12),
  kSCTIMER_BusErrorLFlag,
  kSCTIMER_BusErrorHFlag }
  List of SCTimer flags.

Driver version

- #define FSL_SCTIMER_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  Version 2.1.0.

Initialization and deinitialization

- status_t SCTIMER_Init (SCT_Type ∗base, const sctimer_config_t ∗config)
  Ungates the SCTimer clock and configures the peripheral for basic operation.
- void SCTIMER_Deinit (SCT_Type ∗base)
  Gates the SCTimer clock.
- void SCTIMER_GetDefaultConfig (sctimer_config_t ∗config)

Fills in the SCTimer configuration structure with the default settings.

PWM setup operations

- status_t SCTIMER_SetupPwm (SCT_Type ∗base, const sctimer_pwm_signal_param_t ∗pwmParams, sctimer_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, uint32_t ∗event)
  Configures the PWM signal parameters.
- void SCTIMER_UpdatePwmDutycycle (SCT_Type ∗base, sctimer_out_t output, uint8_t dutyCyclePercent, uint32_t event)
  Updates the duty cycle of an active PWM signal.

Interrupt Interface

- static void SCTIMER_EnableInterrupts (SCT_Type ∗base, uint32_t mask)
  Enables the selected SCTimer interrupts.
- static void SCTIMER_DisableInterrupts (SCT_Type ∗base, uint32_t mask)
  Disables the selected SCTimer interrupts.
- static uint32_t SCTIMER_GetEnabledInterrupts (SCT_Type ∗base)
  Gets the enabled SCTimer interrupts.

Status Interface

- static uint32_t SCTIMER_GetStatusFlags (SCT_Type ∗base)
  Gets the SCTimer status flags.
- static void SCTIMER_ClearStatusFlags (SCT_Type ∗base, uint32_t mask)
  Clears the SCTimer status flags.

Counter Start and Stop

- static void SCTIMER_StartTimer (SCT_Type ∗base, sctimer_counter_t countertoStart)
  Starts the SCTimer counter.
- static void SCTIMER_StopTimer (SCT_Type ∗base, sctimer_counter_t countertoStop)
  Halts the SCTimer counter.

Functions to create a new event and manage the state logic

- status_t SCTIMER_CreateAndScheduleEvent (SCT_Type ∗base, sctimer_event_t howToMonitor, uint32_t matchValue, uint32_t whichIO, sctimer_counter_t whichCounter, uint32_t ∗event)
  Create an event that is triggered on a match or IO and schedule in current state.
- void SCTIMER_ScheduleEvent (SCT_Type ∗base, uint32_t event)
  Enable an event in the current state.
- status_t SCTIMER_IncreaseState (SCT_Type ∗base)
  Increase the state by 1.
- uint32_t SCTIMER_GetCurrentState (SCT_Type ∗base)
  Provides the current state.

Actions to take in response to an event

- status_t SCTIMER_SetupCaptureAction (SCT_Type ∗base, sctimer_counter_t whichCounter, uint32_t ∗captureRegister, uint32_t event)

Setup capture of the counter value on trigger of a selected event.
- void SCTIMER_SetCallback (SCT_Type ∗base, sctimer_event_callback_t callback, uint32_t event)
  Receive noticification when the event trigger an interrupt.
- static void SCTIMER_SetupNextStateAction (SCT_Type ∗base, uint32_t nextState, uint32_t event)
  Transition to the specified state.
- static void SCTIMER_SetupOutputSetAction (SCT_Type ∗base, uint32_t whichIO, uint32_t event)
  Set the Output.
- static void SCTIMER_SetupOutputClearAction (SCT_Type ∗base, uint32_t whichIO, uint32_t event)
  Clear the Output.
- void SCTIMER_SetupOutputToggleAction (SCT_Type ∗base, uint32_t whichIO, uint32_t event)
  Toggle the output level.
- static void SCTIMER_SetupCounterLimitAction (SCT_Type ∗base, sctimer_counter_t whichCounter, uint32_t event)
  Limit the running counter.
- static void SCTIMER_SetupCounterStopAction (SCT_Type ∗base, sctimer_counter_t whichCounter, uint32_t event)
  Stop the running counter.
- static void SCTIMER_SetupCounterStartAction (SCT_Type ∗base, sctimer_counter_t whichCounter, uint32_t event)
  Re-start the stopped counter.
- static void SCTIMER_SetupCounterHaltAction (SCT_Type ∗base, sctimer_counter_t whichCounter, uint32_t event)
  Halt the running counter.
- static void SCTIMER_SetupDmaTriggerAction (SCT_Type ∗base, uint32_t dmaNumber, uint32_t event)
  Generate a DMA request.
- void SCTIMER_EventHandleIRQ (SCT_Type ∗base)
  SCTimer interrupt handler.

23.6 Data Structure Documentation

23.6.1 struct sctimer_pwm_signal_param_t

Data Fields

- sctimer_out_t output
  The output pin to use to generate the PWM signal.
- sctimer_pwm_level_select_t level
  PWM output active level select.
- uint8_t dutyCyclePercent
  PWM pulse width, value should be between 1 to 100 100 = always active signal (100% duty cycle).

23.6.1.0.0.20 Field Documentation

23.6.1.0.0.20.1 sctimer_pwm_level_select_t sctimer_pwm_signal_param_t::level

23.6.1.0.0.20.2 uint8_t sctimer_pwm_signal_param_t::dutyCyclePercent

23.6.2 struct sctimer_config_t

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

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

Data Fields

- bool enableCounterUnify
  true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters
- sctimer_clock_mode_t clockMode
  SCT clock mode value.
- sctimer_clock_select_t clockSelect
  SCT clock select value.
- bool enableBidirection_l
  true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter
- bool enableBidirection_h
  true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter.
- uint8_t prescale_l
  Prescale value to produce the L or unified counter clock.
- uint8_t prescale_h
  Prescale value to produce the H counter clock.
- uint8_t outInitState
  Defines the initial output value.
- uint8_t inputsync
  SCT INSYNC value, INSYNC field in the CONFIG register, from bit9 to bit 16.

23.6.2.0.0.21 Field Documentation

23.6.2.0.0.21.1 bool sctimer_config_t::enableBidirection_h

This field is used only if the enableCounterUnify is set to false

23.6.2.0.0.21.2 uint8_t sctimer_config_t::prescale_h

This field is used only if the enableCounterUnify is set to false

23.6.2.0.0.21.3 uint8_t sctimer_config_t::inputsync

it is used to define synchronization for input N: bit 9 = input 0 bit 10 = input 1 bit 11 = input 2 bit 12 = input 3 All other bits are reserved (bit13 ∼bit 16). How User to set the the value for the member inputsync. IE: delay for input0, and input 1, bypasses for input 2 and input 3 MACRO definition in user level. #define INPUTSYNC0 (0U) #define INPUTSYNC1 (1U) #define INPUTSYNC2 (2U) #define INPUTSYNC3 (3U) User Code. sctimerInfo.inputsync = (1 << INPUTSYNC2) | (1 << INPUTSYNC3);

23.7 Typedef Documentation

23.7.1 typedef void(∗ sctimer_event_callback_t)(void)

23.8 Enumeration Type Documentation

23.8.1 enum sctimer_pwm_mode_t

Enumerator

kSCTIMER_EdgeAlignedPwm Edge-aligned PWM.
kSCTIMER_CenterAlignedPwm Center-aligned PWM.

23.8.2 enum sctimer_counter_t

Enumerator

kSCTIMER_Counter_L Counter L.
kSCTIMER_Counter_H Counter H.

23.8.3 enum sctimer_input_t

Enumerator

kSCTIMER_Input_0 SCTIMER input 0.
kSCTIMER_Input_1 SCTIMER input 1.
kSCTIMER_Input_2 SCTIMER input 2.
kSCTIMER_Input_3 SCTIMER input 3.
kSCTIMER_Input_4 SCTIMER input 4.
kSCTIMER_Input_5 SCTIMER input 5.
kSCTIMER_Input_6 SCTIMER input 6.
kSCTIMER_Input_7 SCTIMER input 7.

23.8.4 enum sctimer_out_t

Enumerator

kSCTIMER_Out_0 SCTIMER output 0.
kSCTIMER_Out_1 SCTIMER output 1.
kSCTIMER_Out_2 SCTIMER output 2.
kSCTIMER_Out_3 SCTIMER output 3.
kSCTIMER_Out_4 SCTIMER output 4.
kSCTIMER_Out_5 SCTIMER output 5.
kSCTIMER_Out_6 SCTIMER output 6.
kSCTIMER_Out_7 SCTIMER output 7.
kSCTIMER_Out_8 SCTIMER output 8.
kSCTIMER_Out_9 SCTIMER output 9.

23.8.5 enum sctimer_pwm_level_select_t

Enumerator

kSCTIMER_LowTrue Low true pulses.
kSCTIMER_HighTrue High true pulses.

23.8.6 enum sctimer_clock_mode_t

Enumerator

kSCTIMER_System_ClockMode System Clock Mode.
kSCTIMER_Sampled_ClockMode Sampled System Clock Mode.
kSCTIMER_Input_ClockMode SCT Input Clock Mode.
kSCTIMER_Asynchronous_ClockMode Asynchronous Mode.

23.8.7 enum sctimer_clock_select_t

Enumerator

kSCTIMER_Clock_On_Rise_Input_0 Rising edges on input 0.
kSCTIMER_Clock_On_Fall_Input_0 Falling edges on input 0.
kSCTIMER_Clock_On_Rise_Input_1 Rising edges on input 1.
kSCTIMER_Clock_On_Fall_Input_1 Falling edges on input 1.
kSCTIMER_Clock_On_Rise_Input_2 Rising edges on input 2.
kSCTIMER_Clock_On_Fall_Input_2 Falling edges on input 2.
kSCTIMER_Clock_On_Rise_Input_3 Rising edges on input 3.

kSCTIMER_Clock_On_Fall_Input_3 Falling edges on input 3.
kSCTIMER_Clock_On_Rise_Input_4 Rising edges on input 4.
kSCTIMER_Clock_On_Fall_Input_4 Falling edges on input 4.
kSCTIMER_Clock_On_Rise_Input_5 Rising edges on input 5.
kSCTIMER_Clock_On_Fall_Input_5 Falling edges on input 5.
kSCTIMER_Clock_On_Rise_Input_6 Rising edges on input 6.
kSCTIMER_Clock_On_Fall_Input_6 Falling edges on input 6.
kSCTIMER_Clock_On_Rise_Input_7 Rising edges on input 7.
kSCTIMER_Clock_On_Fall_Input_7 Falling edges on input 7.

23.8.8 enum sctimer_conflict_resolution_t

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Enumerator

kSCTIMER_ResolveNone No change.
kSCTIMER_ResolveSet Set output.
kSCTIMER_ResolveClear Clear output.
kSCTIMER_ResolveToggle Toggle output.

23.8.9 enum sctimer_interrupt_enable_t

Enumerator

kSCTIMER_Event0InterruptEnable Event 0 interrupt.
kSCTIMER_Event1InterruptEnable Event 1 interrupt.
kSCTIMER_Event2InterruptEnable Event 2 interrupt.
kSCTIMER_Event3InterruptEnable Event 3 interrupt.
kSCTIMER_Event4InterruptEnable Event 4 interrupt.
kSCTIMER_Event5InterruptEnable Event 5 interrupt.
kSCTIMER_Event6InterruptEnable Event 6 interrupt.
kSCTIMER_Event7InterruptEnable Event 7 interrupt.
kSCTIMER_Event8InterruptEnable Event 8 interrupt.
kSCTIMER_Event9InterruptEnable Event 9 interrupt.
kSCTIMER_Event10InterruptEnable Event 10 interrupt.
kSCTIMER_Event11InterruptEnable Event 11 interrupt.
kSCTIMER_Event12InterruptEnable Event 12 interrupt.

23.8.10 enum sctimer_status_flags_t

Enumerator

kSCTIMER_Event0Flag Event 0 Flag.
kSCTIMER_Event1Flag Event 1 Flag.
kSCTIMER_Event2Flag Event 2 Flag.
kSCTIMER_Event3Flag Event 3 Flag.
kSCTIMER_Event4Flag Event 4 Flag.
kSCTIMER_Event5Flag Event 5 Flag.
kSCTIMER_Event6Flag Event 6 Flag.
kSCTIMER_Event7Flag Event 7 Flag.
kSCTIMER_Event8Flag Event 8 Flag.
kSCTIMER_Event9Flag Event 9 Flag.
kSCTIMER_Event10Flag Event 10 Flag.
kSCTIMER_Event11Flag Event 11 Flag.
kSCTIMER_Event12Flag Event 12 Flag.
kSCTIMER_BusErrorLFlag Bus error due to write when L counter was not halted.
kSCTIMER_BusErrorHFlag Bus error due to write when H counter was not halted.

23.9 Function Documentation

23.9.1 status_t SCTIMER_Init (SCT_Type ∗ *base*, const sctimer_config_t ∗ *config*)

Note

This API should be called at the beginning of the application using the SCTimer driver.

Parameters

base	SCTimer peripheral base address
config	Pointer to the user configuration structure.

Returns

kStatus_Success indicates success; Else indicates failure.

23.9.2 void SCTIMER_Deinit (SCT_Type ∗ *base*)

Parameters

base	SCTimer peripheral base address

23.9.3 void SCTIMER_GetDefaultConfig (sctimer_config_t * *config*)

The default values are:

```
*   config->enableCounterUnify = true;
*   config->clockMode = kSCTIMER_System_ClockMode;
*   config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
*   config->enableBidirection_l = false;
*   config->enableBidirection_h = false;
*   config->prescale_l = 0U;
*   config->prescale_h = 0U;
*   config->outInitState = 0U;
*   config->inputsync  = 0xFU;
*
```

Parameters

config	Pointer to the user configuration structure.

23.9.4 status_t SCTIMER_SetupPwm (SCT_Type * *base*, const sctimer_pwm_signal_param_t * *pwmParams*, sctimer_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*, uint32_t * *event*)

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function SCTIMER_GetCurrentStateNumber(). The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

Note

> When setting PWM output from multiple output pins, they all should use the same PWM mode i.e all PWM's should be either edge-aligned or center-aligned. When using this API, the PWM signal frequency of all the initialized channels must be the same. Otherwise all the initialized channels' PWM signal frequency is equal to the last call to the API's pwmFreq_Hz.

Parameters

base	SCTimer peripheral base address
pwmParams	PWM parameters to configure the output
mode	PWM operation mode, options available in enumeration sctimer_pwm_mode_t
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	SCTimer counter clock in Hz
event	Pointer to a variable where the PWM period event number is stored

Returns

kStatus_Success on success kStatus_Fail If we have hit the limit in terms of number of events created or if an incorrect PWM dutycylce is passed in.

23.9.5 void SCTIMER_UpdatePwmDutycycle (SCT_Type ∗ *base*, sctimer_out_t *output*, uint8_t *dutyCyclePercent*, uint32_t *event*)

Parameters

base	SCTimer peripheral base address
output	The output to configure
dutyCycle-Percent	New PWM pulse width; the value should be between 1 to 100
event	Event number associated with this PWM signal. This was returned to the user by the function SCTIMER_SetupPwm().

23.9.6 static void SCTIMER_EnableInterrupts (SCT_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address

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

23.9.7 static void SCTIMER_DisableInterrupts (SCT_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration sctimer-_interrupt_enable_t

23.9.8 static uint32_t SCTIMER_GetEnabledInterrupts (SCT_Type ∗ *base*) [inline], [static]

Parameters

base	SCTimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration sctimer_interrupt_-enable_t

23.9.9 static uint32_t SCTIMER_GetStatusFlags (SCT_Type ∗ *base*) [inline], [static]

Parameters

base	SCTimer peripheral base address

Returns

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

23.9.10 static void SCTIMER_ClearStatusFlags (SCT_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration sctimer-_status_flags_t

23.9.11 static void SCTIMER_StartTimer (SCT_Type * *base*, sctimer_counter_t *countertoStart*) [inline], [static]

Parameters

base	SCTimer peripheral base address
countertoStart	SCTimer counter to start; if unify mode is set then function always writes to HALT_L bit

23.9.12 static void SCTIMER_StopTimer (SCT_Type * *base*, sctimer_counter_t *countertoStop*) [inline], [static]

Parameters

base	SCTimer peripheral base address
countertoStop	SCTimer counter to stop; if unify mode is set then function always writes to HALT_L bit

23.9.13 status_t SCTIMER_CreateAndScheduleEvent (SCT_Type * *base*, sctimer_event_t *howToMonitor*, uint32_t *matchValue*, uint32_t *whichIO*, sctimer_counter_t *whichCounter*, uint32_t * *event*)

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

Parameters

base	SCTimer peripheral base address
howToMonitor	Event type; options are available in the enumeration sctimer_interrupt_enable_t
matchValue	The match value that will be programmed to a match register
whichIO	The input or output that will be involved in event triggering. This field is ignored if the event type is "match only"
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as we have only 1 unified counter; hence ignored.
event	Pointer to a variable where the new event number is stored

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

23.9.14 void SCTIMER_ScheduleEvent (SCT_Type ∗ *base,* uint32_t *event*)

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function SCTIMER_SetupPwm() or function SCTIMER_CreateAndScheduleEvent()
.

Parameters

base	SCTimer peripheral base address
event	Event number to enable in the current state

23.9.15 status_t SCTIMER_IncreaseState (SCT_Type ∗ *base*)

All future events created by calling the function SCTIMER_ScheduleEvent() will be enabled in this new state.

Parameters

base	SCTimer peripheral base address

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of states used

23.9.16 uint32_t SCTIMER_GetCurrentState (SCT_Type * *base*)

User can use this to set the next state by calling the function SCTIMER_SetupNextStateAction().

Parameters

base	SCTimer peripheral base address

Returns

The current state

23.9.17 status_t SCTIMER_SetupCaptureAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t * *captureRegister*, uint32_t *event*)

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
captureRegister	Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
event	Event number that will trigger the capture

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

23.9.18 void SCTIMER_SetCallback (SCT_Type * *base*, sctimer_event_callback_t *callback*, uint32_t *event*)

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

Parameters

base	SCTimer peripheral base address

event	Event number that will trigger the interrupt
callback	Function to invoke when the event is triggered

23.9.19 static void SCTIMER_SetupNextStateAction (SCT_Type * *base,* uint32_t *nextState,* uint32_t *event*) [inline], [static]

This transition will be triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
nextState	The next state SCTimer will transition to
event	Event number that will trigger the state transition

23.9.20 static void SCTIMER_SetupOutputSetAction (SCT_Type * *base,* uint32_t *whichIO,* uint32_t *event*) [inline], [static]

This output will be set when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichIO	The output to set
event	Event number that will trigger the output change

23.9.21 static void SCTIMER_SetupOutputClearAction (SCT_Type * *base,* uint32_t *whichIO,* uint32_t *event*) [inline], [static]

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichIO	The output to clear
event	Event number that will trigger the output change

23.9.22 void SCTIMER_SetupOutputToggleAction (SCT_Type * *base*, uint32_t *whichIO*, uint32_t *event*)

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
whichIO	The output to toggle
event	Event number that will trigger the output change

23.9.23 static void SCTIMER_SetupCounterLimitAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be limited

23.9.24 static void SCTIMER_SetupCounterStopAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be stopped

23.9.25 static void SCTIMER_SetupCounterStartAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to re-start

23.9.26 static void SCTIMER_SetupCounterHaltAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the SCTIMER_StartTimer() function.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be halted

23.9.27 static void SCTIMER_SetupDmaTriggerAction (SCT_Type * *base*, uint32_t *dmaNumber*, uint32_t *event*) [inline], [static]

DMA request will be triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
dmaNumber	The DMA request to generate
event	Event number that will trigger the DMA request

23.9.28 void SCTIMER_EventHandleIRQ (SCT_Type * *base*)

Parameters

base	SCTimer peripheral base address.

Chapter 24
WKT: Self-wake-up Timer

24.1 Overview

The MCUXpresso SDK provides a driver for the Self-wake-up Timer (WKT) of MCUXpresso SDK devices.

24.2 Function groups

The WKT driver supports operating the module as a time counter.

24.2.1 Initialization and deinitialization

The function WKT_Init() initializes the WKT with specified configurations. The function WKT_GetDefaultConfig() gets the default configurations. The initialization function configures the WKT operating mode.

The function WKT_Deinit() stops the WKT timers and disables the module clock.

24.2.2 Read actual WKT counter value

The function WKT_GetCounterValue() reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

24.2.3 Start and Stop timer operations

The function WKT_StartTimer() starts the timer counting. After calling this function, the timer loads the period value, counts down to 0. When the timer reaches 0, it stops and generates a trigger pulse and sets the timeout interrupt flag.

The function WKT_StopTimer() stops the timer counting.

24.2.4 Status

Provides functions to get and clear the WKT status flags.

24.3 Typical use case

24.3.1 WKT tick example

Updates the WKT period and toggles an LED periodically. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/wkt

Files

- file fsl_wkt.h

Data Structures

- struct wkt_config_t
  Describes WKT configuration structure. More...

Enumerations

- enum wkt_clock_source_t {
  kWKT_DividedFROClockSource = 0U,
  kWKT_LowPowerClockSource = 1U,
  kWKT_ExternalClockSource = 2U }
  Describes WKT clock source.
- enum wkt_status_flags_t { kWKT_AlarmFlag = WKT_CTRL_ALARMFLAG_MASK }
  List of WKT flags.

Driver version

- #define FSL_WKT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  Version 2.0.1.

Initialization and deinitialization

- void WKT_Init (WKT_Type ∗base, const wkt_config_t ∗config)
  Ungates the WKT clock and configures the peripheral for basic operation.
- void WKT_Deinit (WKT_Type ∗base)
  Gate the WKT clock.
- static void WKT_GetDefaultConfig (wkt_config_t ∗config)
  Initializes the WKT configuration structure.

Read the counter value.

- static uint32_t WKT_GetCounterValue (WKT_Type ∗base)
  Read actual WKT counter value.

Status Interface

- static uint32_t WKT_GetStatusFlags (WKT_Type ∗base)
  Gets the WKT status flags.
- static void WKT_ClearStatusFlags (WKT_Type ∗base, uint32_t mask)
  Clears the WKT status flags.

Timer Start and Stop

- static void WKT_StartTimer (WKT_Type *base, uint32_t count)
  Starts the timer counting.
- static void WKT_StopTimer (WKT_Type *base)
  Stops the timer counting.

24.4 Data Structure Documentation

24.4.1 struct wkt_config_t

Data Fields

- wkt_clock_source_t clockSource
  External or internal clock source select.

24.5 Enumeration Type Documentation

24.5.1 enum wkt_clock_source_t

Enumerator

kWKT_DividedFROClockSource WKT clock sourced from the divided FRO clock.
kWKT_LowPowerClockSource WKT clock sourced from the Low power clock Use this clock, LPOSCEN bit of DPDCTRL register must be enabled.
kWKT_ExternalClockSource WKT clock sourced from the Low power clock Use this clock, WAKECLKPAD_DISABLE bit of DPDCTRL register must be enabled.

24.5.2 enum wkt_status_flags_t

Enumerator

kWKT_AlarmFlag Alarm flag.

24.6 Function Documentation

24.6.1 void WKT_Init (WKT_Type * *base*, const wkt_config_t * *config*)

Note

This API should be called at the beginning of the application using the WKT driver.

Parameters

base	WKT peripheral base address
config	Pointer to user's WKT config structure.

24.6.2 void WKT_Deinit (WKT_Type * *base*)

Parameters

base	WKT peripheral base address

24.6.3 static void WKT_GetDefaultConfig (wkt_config_t * *config*) [inline], [static]

This function initializes the WKT configuration structure to default values. The default values are as follows.

```
*   config->clockSource = kWKT_DividedFROClockSource;
*
```

Parameters

config	Pointer to the WKT configuration structure.

See Also

wkt_config_t

24.6.4 static uint32_t WKT_GetCounterValue (WKT_Type * *base*) [inline], [static]

Parameters

base	WKT peripheral base address

24.6.5 static uint32_t WKT_GetStatusFlags (WKT_Type * *base*) [inline], [static]

Parameters

base	WKT peripheral base address

Returns

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

24.6.6 static void WKT_ClearStatusFlags (WKT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	WKT peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration wkt_-status_flags_t

24.6.7 static void WKT_StartTimer (WKT_Type * *base*, uint32_t *count*) [inline], [static]

After calling this function, timer loads a count value, counts down to 0, then stops.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks Do not write to Counter register while the counting is in progress

Parameters

base	WKT peripheral base address.
count	The value to be loaded into the WKT Count register

24.6.8 static void WKT_StopTimer (WKT_Type * *base*) [inline], [static]

This function Clears the counter and stops the timer from counting.

Parameters

base	WKT peripheral base address

Chapter 25
WWDT: Windowed Watchdog Timer Driver

25.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Watchdog module (WDOG) of MCUXpresso SDK devices.

25.2 Function groups

25.2.1 Initialization and deinitialization

The function WWDT_Init() initializes the watchdog timer with specified configurations. The configurations include timeout value and whether to enable watchdog after init. The function WWDT_GetDefaultConfig() gets the default configurations.

The function WWDT_Deinit() disables the watchdog and the module clock.

25.2.2 Status

Provides functions to get and clear the WWDT status.

25.2.3 Interrupt

Provides functions to enable/disable WWDT interrupts and get current enabled interrupts.

25.2.4 Watch dog Refresh

The function WWDT_Refresh() feeds the WWDT.

25.3 Typical use case

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

Files

- file fsl_wwdt.h

Data Structures

- struct wwdt_config_t
  Describes WWDT configuration structure. More...

Enumerations

- enum _wwdt_status_flags_t {
kWWDT_TimeoutFlag = WWDT_MOD_WDTOF_MASK,
kWWDT_WarningFlag = WWDT_MOD_WDINT_MASK }
  WWDT status flags.

Driver version

- #define FSL_WWDT_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
  Defines WWDT driver version 2.1.2.

Refresh sequence

- #define WWDT_FIRST_WORD_OF_REFRESH (0xAAU)
  First word of refresh sequence.
- #define WWDT_SECOND_WORD_OF_REFRESH (0x55U)
  Second word of refresh sequence.

WWDT Initialization and De-initialization

- void WWDT_GetDefaultConfig (wwdt_config_t ∗config)
  Initializes WWDT configure structure.
- void WWDT_Init (WWDT_Type ∗base, const wwdt_config_t ∗config)
  Initializes the WWDT.
- void WWDT_Deinit (WWDT_Type ∗base)
  Shuts down the WWDT.

WWDT Functional Operation

- static void WWDT_Enable (WWDT_Type ∗base)
  Enables the WWDT module.
- static void WWDT_Disable (WWDT_Type ∗base)
  Disables the WWDT module.
- static uint32_t WWDT_GetStatusFlags (WWDT_Type ∗base)
  Gets all WWDT status flags.
- void WWDT_ClearStatusFlags (WWDT_Type ∗base, uint32_t mask)
  Clear WWDT flag.
- static void WWDT_SetWarningValue (WWDT_Type ∗base, uint32_t warningValue)
  Set the WWDT warning value.
- static void WWDT_SetTimeoutValue (WWDT_Type ∗base, uint32_t timeoutCount)
  Set the WWDT timeout value.
- static void WWDT_SetWindowValue (WWDT_Type ∗base, uint32_t windowValue)
  Sets the WWDT window value.
- void WWDT_Refresh (WWDT_Type ∗base)
  Refreshes the WWDT timer.

25.4 Data Structure Documentation

25.4.1 struct wwdt_config_t

Data Fields

- bool enableWwdt
  Enables or disables WWDT.
- bool enableWatchdogReset
  true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset
- bool enableWatchdogProtect
  true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time
- bool enableLockOscillator
  true: Disabling or powering down the watchdog oscillator is prevented Once set, this bit can only be cleared by a reset false: Do not lock oscillator
- uint32_t windowValue
  Window value, set this to 0xFFFFFF if windowing is not in effect.
- uint32_t timeoutValue
  Timeout value.
- uint32_t warningValue
  Watchdog time counter value that will generate a warning interrupt.
- uint32_t clockFreq_Hz
  Watchdog clock source frequency.

25.4.1.0.0.22 Field Documentation

25.4.1.0.0.22.1 uint32_t wwdt_config_t::warningValue

Set this to 0 for no warning

25.4.1.0.0.22.2 uint32_t wwdt_config_t::clockFreq_Hz

25.5 Macro Definition Documentation

25.5.1 #define FSL_WWDT_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

25.6 Enumeration Type Documentation

25.6.1 enum _wwdt_status_flags_t

This structure contains the WWDT status flags for use in the WWDT functions.

Enumerator

kWWDT_TimeoutFlag Time-out flag, set when the timer times out.
kWWDT_WarningFlag Warning interrupt flag, set when timer is below the value WDWARNINT.

25.7 Function Documentation

25.7.1 void WWDT_GetDefaultConfig (wwdt_config_t * config)

This function initializes the WWDT configure structure to default value. The default value are:

```
*   config->enableWwdt = true;
*   config->enableWatchdogReset = false;
*   config->enableWatchdogProtect = false;
*   config->enableLockOscillator = false;
*   config->windowValue = 0xFFFFFFU;
*   config->timeoutValue = 0xFFFFFFU;
*   config->warningValue = 0;
*
```

Parameters

config	Pointer to WWDT config structure.

See Also

wwdt_config_t

25.7.2 void WWDT_Init (WWDT_Type * base, const wwdt_config_t * config)

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
*    wwdt_config_t config;
*    WWDT_GetDefaultConfig(&config);
*    config.timeoutValue = 0x7ffU;
*    WWDT_Init(wwdt_base,&config);
*
```

Parameters

base	WWDT peripheral base address
config	The configuration of WWDT

25.7.3 void WWDT_Deinit (WWDT_Type * base)

This function shuts down the WWDT.

Parameters

base	WWDT peripheral base address

25.7.4 static void WWDT_Enable (WWDT_Type * *base*) [inline], [static]

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

base	WWDT peripheral base address

25.7.5 static void WWDT_Disable (WWDT_Type * *base*) [inline], [static]

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

base	WWDT peripheral base address

25.7.6 static uint32_t WWDT_GetStatusFlags (WWDT_Type * *base*) [inline], [static]

This function gets all status flags.

Example for getting Timeout Flag:

```
*   uint32_t status;
*   status = WWDT_GetStatusFlags(wwdt_base) &
      kWWDT_TimeoutFlag;
*
```

Parameters

base	WWDT peripheral base address

Returns

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

25.7.7 void WWDT_ClearStatusFlags (WWDT_Type ∗ *base*, uint32_t *mask*)

This function clears WWDT status flag.

Example for clearing warning flag:

```
*   WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
*
```

Parameters

base	WWDT peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration _wwdt-_status_flags_t

25.7.8 static void WWDT_SetWarningValue (WWDT_Type ∗ *base*, uint32_t *warningValue*) [inline], [static]

The WDWARNINT register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by WARNINT, an interrupt will be generated after the subsequent WDCLK.

Parameters

base	WWDT peripheral base address
warningValue	WWDT warning value.

25.7.9 static void WWDT_SetTimeoutValue (WWDT_Type ∗ *base*, uint32_t *timeoutCount*) [inline], [static]

This function sets the timeout value. Every time a feed sequence occurs the value in the TC register is loaded into the Watchdog timer. Writing a value below 0xFF will cause 0xFF to be loaded into the TC register. Thus the minimum time-out interval is TWDCLK∗256∗4. If enableWatchdogProtect flag is true in wwdt_config_t config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the WDTOF flag.

Parameters

base	WWDT peripheral base address
timeoutCount	WWDT timeout value, count of WWDT clock tick.

25.7.10 static void WWDT_SetWindowValue (WWDT_Type * *base,* uint32_t *windowValue*) [inline], [static]

The WINDOW register determines the highest TV value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in WINDOW, a watchdog event will occur. To disable windowing, set windowValue to 0xFFFFFF (maximum possible timer value) so windowing is not in effect.

Parameters

base	WWDT peripheral base address
windowValue	WWDT window value.

25.7.11 void WWDT_Refresh (WWDT_Type * *base*)

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

base	WWDT peripheral base address

Chapter 26
SPI: Serial Peripheral Interface Driver

26.1 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 the purpose of optimization/customization. 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 API 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_SlaveTransferCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions SPI_MasterTransferNonBlocking() 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.

26.2 Typical use case

26.2.1 SPI master transfer using an interrupt method

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

Modules

- SPI Driver

26.3 SPI Driver

26.3.1 Overview

This section describes the programming interface of the SPI driver.

Files

- file fsl_spi.h

Data Structures

- struct spi_delay_config_t
    SPI delay time configure structure. More...
- 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 (0xFFFFU)
    SPI dummy transfer data, the data is sent while txBuff is NULL.

Typedefs

- typedef spi_master_handle_t spi_slave_handle_t
    Slave handle type.
- typedef void(∗ spi_master_callback_t)(SPI_Type ∗base, spi_master_handle_t ∗handle, status_t status, void ∗userData)
    SPI master callback for finished transmit.
- typedef void(∗ spi_slave_callback_t)(SPI_Type ∗base, spi_slave_handle_t ∗handle, status_t status, void ∗userData)
    SPI slave callback for finished transmit.

Enumerations

- enum _spi_xfer_option {
  kSPI_EndOfFrame = (SPI_TXDATCTL_EOF_MASK),
  kSPI_EndOfTransfer = (SPI_TXDATCTL_EOT_MASK),

kSPI_ReceiveIgnore = (SPI_TXDATCTL_RXIGNORE_MASK) }
SPI transfer option.
- enum spi_shift_direction_t {
kSPI_MsbFirst = 0U,
kSPI_LsbFirst = 1U }
SPI data shifter direction options.
- enum spi_clock_polarity_t {
kSPI_ClockPolarityActiveHigh = 0x0U,
kSPI_ClockPolarityActiveLow = 0x1U }
SPI clock polarity configuration.
- enum spi_clock_phase_t {
kSPI_ClockPhaseFirstEdge = 0x0U,
kSPI_ClockPhaseSecondEdge = 0x1U }
SPI clock phase configuration.
- enum spi_ssel_t { kSPI_Ssel0Assert = (int)(∼SPI_TXDATCTL_TXSSEL0_N_MASK) }
Slave select.
- enum spi_spol_t
ssel polarity
- enum _spi_status {
kStatus_SPI_Busy = MAKE_STATUS(kStatusGroup_LPC_MINISPI, 0),
kStatus_SPI_Idle = MAKE_STATUS(kStatusGroup_LPC_MINISPI, 1),
kStatus_SPI_Error = MAKE_STATUS(kStatusGroup_LPC_MINISPI, 2),
kStatus_SPI_BaudrateNotSupport }
SPI transfer status.
- enum _spi_interrupt_enable {
kSPI_RxReadyInterruptEnable = SPI_INTENSET_RXRDYEN_MASK,
kSPI_TxReadyInterruptEnable = SPI_INTENSET_TXRDYEN_MASK,
kSPI_RxOverrunInterruptEnable = SPI_INTENSET_RXOVEN_MASK,
kSPI_TxUnderrunInterruptEnable = SPI_INTENSET_TXUREN_MASK,
kSPI_SlaveSelectAssertInterruptEnable = SPI_INTENSET_SSAEN_MASK,
kSPI_SlaveSelectDeassertInterruptEnable = SPI_INTENSET_SSDEN_MASK }
SPI interrupt sources.
- enum _spi_status_flags {
kSPI_RxReadyFlag = SPI_STAT_RXRDY_MASK,
kSPI_TxReadyFlag = SPI_STAT_TXRDY_MASK,
kSPI_RxOverrunFlag = SPI_STAT_RXOV_MASK,
kSPI_TxUnderrunFlag = SPI_STAT_TXUR_MASK,
kSPI_SlaveSelectAssertFlag = SPI_STAT_SSA_MASK,
kSPI_SlaveSelectDeassertFlag = SPI_STAT_SSD_MASK,
kSPI_StallFlag = SPI_STAT_STALLED_MASK,
kSPI_EndTransferFlag = SPI_STAT_ENDTRANSFER_MASK,
kSPI_MasterIdleFlag = SPI_STAT_MSTIDLE_MASK }
SPI status flags.

Functions

- uint32_t SPI_GetInstance (SPI_Type ∗base)
  Returns instance number for SPI peripheral base address.

Driver version

- #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  SPI driver version 2.0.1.

Initialization and deinitialization

- void SPI_MasterGetDefaultConfig (spi_master_config_t ∗config)
  Sets the SPI master configuration structure to default values.
- status_t 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.
- status_t 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)
  Enable or disable the SPI Master or Slave.

Status

- static uint32_t SPI_GetStatusFlags (SPI_Type ∗base)
  Gets the status flag.
- static void SPI_ClearStatusFlags (SPI_Type ∗base, uint32_t mask)
  Clear the status flag.

Interrupts

- static void SPI_EnableInterrupts (SPI_Type ∗base, uint32_t irqs)
  Enables the interrupt for the SPI.
- static void SPI_DisableInterrupts (SPI_Type ∗base, uint32_t irqs)
  Disables the interrupt for the SPI.

Bus Operations

- static bool SPI_IsMaster (SPI_Type ∗base)
  Returns whether the SPI module is in master mode.

- status_t SPI_MasterSetBaudRate (SPI_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
  Sets the baud rate for SPI transfer.
- static void SPI_WriteData (SPI_Type *base, uint16_t data)
  Writes a data into the SPI data register directly.
- static void SPI_WriteConfigFlags (SPI_Type *base, uint32_t configFlags)
  Writes a data into the SPI TXCTL register directly.
- void SPI_WriteDataWithConfigFlags (SPI_Type *base, uint16_t data, uint32_t configFlags)
  Writes a data control info and data into the SPI TX register directly.
- static uint32_t SPI_ReadData (SPI_Type *base)
  Gets a data from the SPI data register.
- void SPI_SetTransferDelay (SPI_Type *base, const spi_delay_config_t *config)
  Set delay time for transfer.
- void SPI_SetDummyData (SPI_Type *base, uint16_t dummyData)
  Set up the dummy data.
- status_t SPI_MasterTransferBlocking (SPI_Type *base, spi_transfer_t *xfer)
  Transfers a block of data using a polling method.

Transactional

- status_t 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_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 master transfer count.
- void SPI_MasterTransferAbort (SPI_Type *base, spi_master_handle_t *handle)
  SPI master aborts a transfer using an interrupt.
- void SPI_MasterTransferHandleIRQ (SPI_Type *base, spi_master_handle_t *handle)
  Interrupts the handler for the SPI.
- status_t 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 slave transfer count.
- static void SPI_SlaveTransferAbort (SPI_Type *base, spi_slave_handle_t *handle)
  SPI slave aborts a transfer using an interrupt.
- void SPI_SlaveTransferHandleIRQ (SPI_Type *base, spi_slave_handle_t *handle)
  Interrupts a handler for the SPI slave.

26.3.2 Data Structure Documentation

26.3.2.1 struct spi_delay_config_t

Data Fields

- uint8_t preDelay
  Delay between SSEL assertion and the beginning of transfer.
- uint8_t postDelay
  Delay between the end of transfer and SSEL deassertion.
- uint8_t frameDelay
  Delay between frame to frame.
- uint8_t transferDelay
  Delay between transfer to transfer.

26.3.2.1.0.23 Field Documentation

26.3.2.1.0.23.1 uint8_t spi_delay_config_t::preDelay

26.3.2.1.0.23.2 uint8_t spi_delay_config_t::postDelay

26.3.2.1.0.23.3 uint8_t spi_delay_config_t::frameDelay

26.3.2.1.0.23.4 uint8_t spi_delay_config_t::transferDelay

26.3.2.2 struct spi_master_config_t

Data Fields

- bool enableLoopback
  Enable loopback for test purpose.
- bool enableMaster
  Enable SPI at initialization time.
- uint32_t baudRate_Bps
  Baud Rate for SPI in Hz.
- spi_clock_polarity_t clockPolarity
  Clock polarity.
- spi_clock_phase_t clockPhase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.
- uint8_t dataWidth
  Width of the data.
- spi_ssel_t sselNumber
  Slave select number.
- spi_spol_t sselPolarity
  Configure active CS polarity.
- spi_delay_config_t delayConfig
  Configure for delay time.

26.3.2.2.0.24 Field Documentation

26.3.2.2.0.24.1 spi_delay_config_t spi_master_config_t::delayConfig

26.3.2.3 struct spi_slave_config_t

Data Fields

- bool enableSlave
  Enable SPI at initialization time.
- spi_clock_polarity_t clockPolarity
  Clock polarity.
- spi_clock_phase_t clockPhase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.
- uint8_t dataWidth
  Width of the data.
- spi_spol_t sselPolarity
  Configure active CS polarity.

26.3.2.4 struct spi_transfer_t

Data Fields

- uint8_t * txData
  Send buffer.
- uint8_t * rxData
  Receive buffer.
- size_t dataSize
  Transfer bytes.
- uint32_t configFlags
  Additional option to control transfer _spi_xfer_option.

26.3.2.4.0.25 Field Documentation

26.3.2.4.0.25.1 uint32_t spi_transfer_t::configFlags

26.3.2.5 struct _spi_master_handle

Master handle type.

Data Fields

- uint8_t *volatile txData
  Transfer buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t txRemainingBytes

Number of data to be transmitted [in bytes].
- volatile size_t rxRemainingBytes
  Number of data to be received [in bytes].
- size_t totalByteCount
  A number of transfer bytes.
- volatile uint32_t state
  SPI internal state.
- spi_master_callback_t callback
  SPI callback.
- void * userData
  Callback parameter.
- uint8_t dataWidth
  Width of the data [Valid values: 1 to 16].
- uint32_t lastCommand
  Last command for transfer.

26.3.2.5.0.26 Field Documentation

26.3.2.5.0.26.1 uint32_t spi_master_handle_t::lastCommand

26.3.3 Macro Definition Documentation

26.3.3.1 #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

26.3.3.2 #define SPI_DUMMYDATA (0xFFFFU)

26.3.4 Enumeration Type Documentation

26.3.4.1 enum _spi_xfer_option

Enumerator

kSPI_EndOfFrame Data is treated as the end of a frame.
kSPI_EndOfTransfer Data is treated as the end of a transfer.
kSPI_ReceiveIgnore Ignore the receive data.

26.3.4.2 enum spi_shift_direction_t

Enumerator

kSPI_MsbFirst Data transfers start with most significant bit.
kSPI_LsbFirst Data transfers start with least significant bit.

26.3.4.3 enum spi_clock_polarity_t

Enumerator

kSPI_ClockPolarityActiveHigh Active-high SPI clock (idles low).
kSPI_ClockPolarityActiveLow Active-low SPI clock (idles high).

26.3.4.4 enum spi_clock_phase_t

Enumerator

kSPI_ClockPhaseFirstEdge First edge on SCK occurs at the middle of the first cycle of a data transfer.
kSPI_ClockPhaseSecondEdge First edge on SCK occurs at the start of the first cycle of a data transfer.

26.3.4.5 enum spi_ssel_t

Enumerator

kSPI_Ssel0Assert Slave select 0.

26.3.4.6 enum _spi_status

Enumerator

kStatus_SPI_Busy SPI bus is busy.
kStatus_SPI_Idle SPI is idle.
kStatus_SPI_Error SPI error.
kStatus_SPI_BaudrateNotSupport Baudrate is not support in current clock source.

26.3.4.7 enum _spi_interrupt_enable

Enumerator

kSPI_RxReadyInterruptEnable Rx ready interrupt.
kSPI_TxReadyInterruptEnable Tx ready interrupt.
kSPI_RxOverrunInterruptEnable Rx overrun interrupt.
kSPI_TxUnderrunInterruptEnable Tx underrun interrupt.
kSPI_SlaveSelectAssertInterruptEnable Slave select assert interrupt.
kSPI_SlaveSelectDeassertInterruptEnable Slave select deassert interrupt.

26.3.4.8 enum _spi_status_flags

Enumerator

kSPI_RxReadyFlag Receive ready flag.
kSPI_TxReadyFlag Transmit ready flag.
kSPI_RxOverrunFlag Receive overrun flag.
kSPI_TxUnderrunFlag Transmit underrun flag.
kSPI_SlaveSelectAssertFlag Slave select assert flag.
kSPI_SlaveSelectDeassertFlag slave select deassert flag.
kSPI_StallFlag Stall flag.
kSPI_EndTransferFlag End transfer bit.
kSPI_MasterIdleFlag Master in idle status flag.

26.3.5 Function Documentation

26.3.5.1 uint32_t SPI_GetInstance (SPI_Type ∗ *base*)

26.3.5.2 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

26.3.5.3 status_t 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_MasterGetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
.baudRate_Bps = 500000,
...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

base	SPI base pointer
config	pointer to master configuration structure
srcClock_Hz	Source clock frequency.

26.3.5.4 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

26.3.5.5 status_t 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 = kSPI_ClockPolarityActiveHigh;
.phase = kSPI_ClockPhaseFirstEdge;
.direction = kSPI_MsbFirst;
...
};
SPI_SlaveInit(SPI0, &config);
```

Parameters

base	SPI base pointer
config	pointer to slave configuration structure

26.3.5.6 void SPI_Deinit (SPI_Type * *base*)

Calling this API resets the SPI module, gates the SPI clock. Disable the fifo if enabled. The SPI module can't work unless calling the SPI_MasterInit/SPI_SlaveInit to initialize module.

Parameters

base	SPI base pointer

26.3.5.7 static void SPI_Enable (SPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SPI base pointer
enable	or disable (true = enable, false = disable)

26.3.5.8 static uint32_t SPI_GetStatusFlags (SPI_Type * *base*) [inline], [static]

Parameters

base	SPI base pointer

Returns

SPI Status, use status flag to AND _spi_status_flags could get the related status.

26.3.5.9 static void SPI_ClearStatusFlags (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SPI base pointer
mask	SPI Status, use status flag to AND _spi_status_flags could get the related status.

26.3.5.10 static void SPI_EnableInterrupts (SPI_Type * *base*, uint32_t *irqs*) [inline], [static]

Parameters

base	SPI base pointer
irqs	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxReadyInterruptEnable • kSPI_TxReadyInterruptEnable

26.3.5.11 static void SPI_DisableInterrupts (SPI_Type ∗ *base*, uint32_t *irqs*) [inline], [static]

Parameters

base	SPI base pointer
irqs	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxReadyInterruptEnable • kSPI_TxReadyInterruptEnable

26.3.5.12 static bool SPI_IsMaster (SPI_Type ∗ *base*) [inline], [static]

Parameters

base	SPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

26.3.5.13 status_t 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.

26.3.5.14 static void SPI_WriteData (SPI_Type * *base*, uint16_t *data*) [inline], [static]

Parameters

base	SPI base pointer
data	needs to be write.

26.3.5.15 static void SPI_WriteConfigFlags (SPI_Type * *base*, uint32_t *configFlags*) [inline], [static]

Parameters

base	SPI base pointer
configFlags	control command needs to be write.

26.3.5.16 void SPI_WriteDataWithConfigFlags (SPI_Type * *base*, uint16_t *data*, uint32_t *configFlags*)

Parameters

base	SPI base pointer
value	needs to be write.

26.3.5.17 static uint32_t SPI_ReadData (SPI_Type * *base*) [inline], [static]

Parameters

base	SPI base pointer

Returns

Data in the register.

26.3.5.18 void SPI_SetTransferDelay (SPI_Type * *base*, const spi_delay_config_t * *config*)

```
the delay uint is SPI clock time, maximum value is 0xF.
```

Parameters

base	SPI base pointer
config	configuration for delay option spi_delay_config_t.

26.3.5.19 void SPI_SetDummyData (SPI_Type * *base*, uint16_t *dummyData*)

This API can change the default data to be transferred when users set the tx buffer to NULL.

Parameters

base	SPI peripheral address.
dummyData	Data to be transferred when tx buffer is NULL.

26.3.5.20 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.

26.3.5.21 status_t 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.

26.3.5.22 status_t SPI_MasterTransferNonBlocking (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_transfer_t * *xfer*)

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.

26.3.5.23 status_t SPI_MasterTransferGetCount (SPI_Type * *base*, spi_master_handle_t * *handle*, size_t * *count*)

This function gets the master transfer count.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_master_handle_t structure which stores the transfer state.
count	The number of bytes transferred by using the non-blocking transaction.

Returns

status of status_t.

26.3.5.24 void SPI_MasterTransferAbort (SPI_Type * *base*, spi_master_handle_t * *handle*)

This function aborts a transfer using an interrupt.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_master_handle_t structure which stores the transfer state.

26.3.5.25 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.

26.3.5.26 status_t 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.

26.3.5.27 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.

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.

26.3.5.28 static status_t SPI_SlaveTransferGetCount (SPI_Type * *base*, spi_slave_handle_t * *handle*, size_t * *count*) [inline], [static]

This function gets the slave transfer count.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_master_handle_t structure which stores the transfer state.
count	The number of bytes transferred by using the non-blocking transaction.

Returns

status of status_t.

26.3.5.29 static void SPI_SlaveTransferAbort (SPI_Type * *base*, spi_slave_handle_t * *handle*) [inline], [static]

This function aborts a transfer using an interrupt.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_slave_handle_t structure which stores the transfer state.

26.3.5.30 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

Chapter 27
Debug Console

27.1 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.

27.2 Function groups

27.2.1 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);
```

27.2.2 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 ∗

specifier	Qualifying Input	Type of argument
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(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE    /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF                 DbgConsole_Printf
#define SCANF                  DbgConsole_Scanf
#define PUTCHAR                DbgConsole_Putchar
#define GETCHAR                DbgConsole_Getchar
#else                   /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF                 printf
#define SCANF                  scanf
#define PUTCHAR                putchar
#define GETCHAR                getchar
#endif /* SDK_DEBUGCONSOLE */
```

27.3 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

27.4 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.

27.4.1 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. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

27.4.2 Guide Semihosting for Keil µVision

NOTE: Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

27.4.3 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_DEBU-

G} -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

(a) 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
```

(b) After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 28
Notification Framework

28.1 Overview

This section describes the programming interface of the Notifier driver.

28.2 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.

28.3 Data Structure Documentation

28.3.1 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.

28.3.1.0.0.27 Field Documentation

28.3.1.0.0.27.1 notifier_user_config_t* notifier_notification_block_t::targetConfig

28.3.1.0.0.27.2 notifier_policy_t notifier_notification_block_t::policy

28.3.1.0.0.27.3 notifier_notification_type_t notifier_notification_block_t::notifyType

28.3.2 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.

28.3.2.0.0.28 Field Documentation

28.3.2.0.0.28.1 notifier_callback_t notifier_callback_config_t::callback

28.3.2.0.0.28.2 notifier_callback_type_t notifier_callback_config_t::callbackType

28.3.2.0.0.28.3 void* notifier_callback_config_t::callbackData

28.3.3 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.

28.3.3.0.0.29 Field Documentation

28.3.3.0.0.29.1 notifier_user_config_t** notifier_handle_t::configsTable

28.3.3.0.0.29.2 uint8_t notifier_handle_t::configsNumber

28.3.3.0.0.29.3 notifier_callback_config_t* notifier_handle_t::callbacksTable

28.3.3.0.0.29.4 uint8_t notifier_handle_t::callbacksNumber

28.3.3.0.0.29.5 uint8_t notifier_handle_t::errorCallbackIndex

28.3.3.0.0.29.6 uint8_t notifier_handle_t::currentConfigIndex

28.3.3.0.0.29.7 notifier_user_function_t notifier_handle_t::userFunction

28.3.3.0.0.29.8 void* notifier_handle_t::userData

28.4 Typedef Documentation

28.4.1 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.

28.4.2 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.

28.4.3 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.

28.5 Enumeration Type Documentation

28.5.1 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.

28.5.2 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.

28.5.3 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.

28.5.4 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.

28.6 Function Documentation

28.6.1 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.

28.6.2 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_GetErrorCallback() 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.

28.6.3 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 29
Shell

29.1 Overview

This part describes the programming interface of the Shell middleware. Shell controls MCUs by commands via the specified communication peripheral based on the debug console driver.

29.2 Function groups

29.2.1 Initialization

To initialize the Shell middleware, call the SHELL_Init() function with these parameters. This function automatically enables the middleware.

```
void SHELL_Init(p_shell_context_t context, send_data_cb_t send_cb, recv_data_cb_t recv_cb, char *
      prompt);
```

Then, after the initialization was successful, call a command to control MCUs.

This example shows how to call the SHELL_Init() given the user configuration structure.

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
```

29.2.2 Advanced Feature

- Support to get a character from standard input devices.

```
static uint8_t GetChar(p_shell_context_t context);
```

Commands	Description
Help	Lists all commands which are supported by Shell.
Exit	Exits the Shell program.
strCompare	Compares the two input strings.

Input character	Description
A	Gets the latest command in the history.
B	Gets the first command in the history.
C	Replaces one character at the right of the pointer.

Input character	Description
D	Replaces one character at the left of the pointer.
	Run AutoComplete function
	Run cmdProcess function
	Clears a command.

29.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
SHELL_Main(&user_context);
```

Data Structures

- struct shell_command_t
  User command data configuration structure. More...

Macros

- #define SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE
  Whether use non-blocking mode.
- #define SHELL_AUTO_COMPLETE (1U)
  Macro to set on/off auto-complete feature.
- #define SHELL_BUFFER_SIZE (64U)
  Macro to set console buffer size.
- #define SHELL_MAX_ARGS (8U)
  Macro to set maximum arguments in command.
- #define SHELL_HISTORY_COUNT (3U)
  Macro to set maximum count of history commands.
- #define SHELL_IGNORE_PARAMETER_COUNT (0xFF)
  Macro to bypass arguments check.
- #define SHELL_HANDLE_SIZE (520U)
  The handle size of the shell module.
- #define SHELL_COMMAND_DEFINE(command, descriptor, callback, paramCount)
  Defines the shell command structure.
- #define SHELL_COMMAND(command) &g_shellCommand##command
  Gets the shell command pointer.

Typedefs

- typedef void ∗ shell_handle_t
  The handle of the shell module.
- typedef shell_status_t(∗ cmd_function_t)(shell_handle_t shellHandle, int32_t argc, char ∗∗argv)
  User command function prototype.

Enumerations

- enum shell_status_t {
kStatus_SHELL_Success = kStatus_Success,
kStatus_SHELL_Error = MAKE_STATUS(kStatusGroup_SHELL, 1),
kStatus_SHELL_OpenWriteHandleFailed = MAKE_STATUS(kStatusGroup_SHELL, 2),
kStatus_SHELL_OpenReadHandleFailed = MAKE_STATUS(kStatusGroup_SHELL, 3) }

Shell functional operation

- shell_status_t SHELL_Init (shell_handle_t shellHandle, serial_handle_t serialHandle, char ∗prompt)
  Initializes the shell module.
- shell_status_t SHELL_RegisterCommand (shell_handle_t shellHandle, shell_command_t ∗shellCommand)
  Registers the shell command.
- shell_status_t SHELL_UnregisterCommand (shell_command_t ∗shellCommand)
  Unregisters the shell command.
- shell_status_t SHELL_Write (shell_handle_t shellHandle, char ∗buffer, uint32_t length)
  Sends data to the shell output stream.
- int SHELL_Printf (shell_handle_t shellHandle, const char ∗formatString,...)
  Writes formatted output to the shell output stream.
- void SHELL_Task (shell_handle_t shellHandle)
  The task function for Shell.

29.3 Data Structure Documentation

29.3.1 struct shell_command_t

Data Fields

- const char ∗ pcCommand
  The command that is executed.
- char ∗ pcHelpString
  String that describes how to use the command.
- const cmd_function_t pFuncCallBack
  A pointer to the callback function that returns the output generated by the command.
- uint8_t cExpectedNumberOfParameters
  Commands expect a fixed number of parameters, which may be zero.
- list_element_t link
  link of the element

29.3.1.0.0.30 Field Documentation

29.3.1.0.0.30.1 const char∗ shell_command_t::pcCommand

For example "help". It must be all lower case.

29.3.1.0.0.30.2 char∗ shell_command_t::pcHelpString

It should start with the command itself, and end with "\r\n". For example "help: Returns a list of all the commands\r\n".

29.3.1.0.0.30.3 const cmd_function_t shell_command_t::pFuncCallBack

29.3.1.0.0.30.4 uint8_t shell_command_t::cExpectedNumberOfParameters

29.4 Macro Definition Documentation

29.4.1 #define SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE

29.4.2 #define SHELL_AUTO_COMPLETE (1U)

29.4.3 #define SHELL_BUFFER_SIZE (64U)

29.4.4 #define SHELL_MAX_ARGS (8U)

29.4.5 #define SHELL_HISTORY_COUNT (3U)

29.4.6 #define SHELL_HANDLE_SIZE (520U)

It is the sum of the SHELL_HISTORY_COUNT ∗ SHELL_BUFFER_SIZE + SHELL_BUFFER_SIZE + SERIAL_MANAGER_READ_HANDLE_SIZE + SERIAL_MANAGER_WRITE_HANDLE_SIZE

29.4.7 #define SHELL_COMMAND_DEFINE(*command*, *descriptor*, *callback*, *paramCount*)

Value:

```
\
    shell_command_t g_shellCommand##command = {                         \
        (#command), (descriptor), (callback), (paramCount), {0},        \
    }
```

This macro is used to define the shell command structure shell_command_t. And then uses the macro SHELL_COMMAND to get the command structure pointer. The macro should not be used in any function.

This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);
* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));
*
```

Parameters

command	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".
descriptor	The description of the command is used for showing the command usage when "help" is typing.
callback	The callback of the command is used to handle the command line when the input command is matched.
paramCount	The max parameter count of the current command.

29.4.8 #define SHELL_COMMAND(*command*) &g_shellCommand##command

This macro is used to get the shell command pointer. The macro should not be used before the macro SHELL_COMMAND_DEFINE is used.

Parameters

command	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".

29.5 Typedef Documentation

29.5.1 typedef shell_status_t(* cmd_function_t)(shell_handle_t shellHandle, int32_t argc, char **argv)

29.6 Enumeration Type Documentation

29.6.1 enum shell_status_t

Enumerator

kStatus_SHELL_Success Success.
kStatus_SHELL_Error Failed.
kStatus_SHELL_OpenWriteHandleFailed Open write handle failed.
kStatus_SHELL_OpenReadHandleFailed Open read handle failed.

29.7 Function Documentation

29.7.1 shell_status_t SHELL_Init (shell_handle_t *shellHandle,* serial_handle_t *serialHandle,* char * *prompt*)

This function must be called before calling all other Shell functions. Call operation the Shell commands with user-defined settings. The example below shows how to set up the Shell and how to call the SHELL-_Init function by passing in these parameters. This is an example.

```
*    static uint8_t s_shellHandleBuffer[SHELL_HANDLE_SIZE];
*    static shell_handle_t s_shellHandle = &s_shellHandleBuffer[0];
*    SHELL_Init(s_shellHandle, s_serialHandle, "Test@SHELL>");
*
```

Parameters

shellHandle	Pointer to point to a memory space of size SHELL_HANDLE_SIZE allocated by the caller.
serialHandle	The serial manager module handle pointer.
prompt	The string prompt pointer of Shell. Only the global variable can be passed.

Return values

kStatus_SHELL_Success	The shell initialization succeed.
kStatus_SHELL_Error	An error occurred when the shell is initialized.
kStatus_SHELL_Open-WriteHandleFailed	Open the write handle failed.
kStatus_SHELL_Open-ReadHandleFailed	Open the read handle failed.

29.7.2 shell_status_t SHELL_RegisterCommand (shell_handle_t *shellHandle*, shell_command_t ∗ *shellCommand*)

This function is used to register the shell command by using the command configuration #shell_command-_config_t. This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);
* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));
*
```

Parameters

shellHandle	The shell module handle pointer.
command	The command element.

Return values

kStatus_SHELL_Success	Successfully register the command.
kStatus_SHELL_Error	An error occurred.

29.7.3 shell_status_t SHELL_UnregisterCommand (shell_command_t * *shellCommand*)

This function is used to unregister the shell command.

Parameters

command	The command element.

Return values

kStatus_SHELL_Success	Successfully unregister the command.

29.7.4 shell_status_t SHELL_Write (shell_handle_t *shellHandle,* char * *buffer,* uint32_t *length*)

This function is used to send data to the shell output stream.

Parameters

shellHandle	The shell module handle pointer.
buffer	Start address of the data to write.
length	Length of the data to write.

Return values

kStatus_SHELL_Success	Successfully send data.
kStatus_SHELL_Error	An error occurred.

29.7.5 int SHELL_Printf (shell_handle_t *shellHandle,* const char * *formatString,* ...)

Call this function to write a formatted output to the shell output stream.

Parameters

shellHandle	The shell module handle pointer.
formatString	Format string.

Returns

Returns the number of characters printed or a negative value if an error occurs.

29.7.6 void SHELL_Task (shell_handle_t *shellHandle*)

The task function for Shell; The function should be polled by upper layer. This function does not return until Shell command exit was called.

Parameters

shellHandle	The shell module handle pointer.

Chapter 30
Serial Manager

30.1 Overview

This chapter describes the programming interface of the serial manager component.

The serial manager component provides a series of APIs to operate different serial port types. The port types it supports are UART, USB CDC and SWO.

Modules

- Serial Port SWO
- Serial Port USB
- Serial Port Uart
- Serial Port Virtual USB

Data Structures

- struct serial_manager_config_t
  serial manager config structure More...
- struct serial_manager_callback_message_t
  Callback message structure. More...

Macros

- #define SERIAL_PORT_TYPE_UART (1U)
  Enable or disable uart port (1 - enable, 0 - disable)
- #define SERIAL_PORT_TYPE_USBCDC (0U)
  Enable or disable USB CDC port (1 - enable, 0 - disable)
- #define SERIAL_PORT_TYPE_SWO (0U)
  Enable or disable SWO port (1 - enable, 0 - disable)
- #define SERIAL_PORT_TYPE_USBCDC_VIRTUAL (0U)
  Enable or disable USB CDC virtual port (1 - enable, 0 - disable)
- #define SERIAL_MANAGER_WRITE_HANDLE_SIZE (4U)
  Set serial manager write handle size.
- #define SERIAL_MANAGER_HANDLE_SIZE (SERIAL_MANAGER_HANDLE_SIZE_TEMP + 12U)
  SERIAL_PORT_UART_HANDLE_SIZE/SERIAL_PORT_USB_CDC_HANDLE_SIZE + serial manager dedicated size.

Typedefs

- typedef void(∗ serial_manager_callback_t)(void ∗callbackParam, serial_manager_callback_message_t ∗message, serial_manager_status_t status)
  callback function

Enumerations

- enum serial_port_type_t {
kSerialPort_None = 0U,
kSerialPort_Uart = 1U,
kSerialPort_Uart = 1U,
kSerialPort_UsbCdc,
kSerialPort_Swo,
kSerialPort_UsbCdcVirtual }
  serial port type
- enum serial_manager_status_t {
kStatus_SerialManager_Success = kStatus_Success,
kStatus_SerialManager_Error = MAKE_STATUS(kStatusGroup_SERIALMANAGER, 1),
kStatus_SerialManager_Busy = MAKE_STATUS(kStatusGroup_SERIALMANAGER, 2),
kStatus_SerialManager_Notify = MAKE_STATUS(kStatusGroup_SERIALMANAGER, 3),
kStatus_SerialManager_Canceled,
kStatus_SerialManager_HandleConflict = MAKE_STATUS(kStatusGroup_SERIALMANAGER, 5),
kStatus_SerialManager_RingBufferOverflow }
  serial manager error code

Functions

- serial_manager_status_t SerialManager_Init (serial_handle_t serialHandle, serial_manager_config-_t *config)
  Initializes a serial manager module with the serial manager handle and the user configuration structure.
- serial_manager_status_t SerialManager_Deinit (serial_handle_t serialHandle)
  De-initializes the serial manager module instance.
- serial_manager_status_t SerialManager_OpenWriteHandle (serial_handle_t serialHandle, serial_-write_handle_t writeHandle)
  Opens a writing handle for the serial manager module.
- serial_manager_status_t SerialManager_CloseWriteHandle (serial_write_handle_t writeHandle)
  Closes a writing handle for the serial manager module.
- serial_manager_status_t SerialManager_OpenReadHandle (serial_handle_t serialHandle, serial_-read_handle_t readHandle)
  Opens a reading handle for the serial manager module.
- serial_manager_status_t SerialManager_CloseReadHandle (serial_read_handle_t readHandle)
  Closes a reading for the serial manager module.
- serial_manager_status_t SerialManager_WriteBlocking (serial_write_handle_t writeHandle, uint8-_t *buffer, uint32_t length)
  Transmits data with the blocking mode.
- serial_manager_status_t SerialManager_ReadBlocking (serial_read_handle_t readHandle, uint8_t *buffer, uint32_t length)
  Reads data with the blocking mode.
- serial_manager_status_t SerialManager_EnterLowpower (serial_handle_t serialHandle)
  Prepares to enter low power consumption.
- serial_manager_status_t SerialManager_ExitLowpower (serial_handle_t serialHandle)
  Restores from low power consumption.

30.2 Data Structure Documentation

30.2.1 struct serial_manager_config_t

Data Fields

- uint8_t * ringBuffer
  Ring buffer address, it is used to buffer data received by the hardware.
- uint32_t ringBufferSize
  The size of the ring buffer.
- serial_port_type_t type
  Serial port type.
- void * portConfig
  Serial port configuration.

30.2.1.0.0.31 Field Documentation

30.2.1.0.0.31.1 uint8_t* serial_manager_config_t::ringBuffer

Besides, the memory space cannot be free during the lifetime of the serial manager module.

30.2.2 struct serial_manager_callback_message_t

Data Fields

- uint8_t * buffer
  Transferred buffer.
- uint32_t length
  Transferred data length.

30.3 Enumeration Type Documentation

30.3.1 enum serial_port_type_t

Enumerator

kSerialPort_None Serial port is none.
kSerialPort_Uart Serial port UART.
kSerialPort_Uart Serial port UART.
kSerialPort_UsbCdc Serial port USB CDC.
kSerialPort_Swo Serial port SWO.
kSerialPort_UsbCdcVirtual Serial port USB CDC Virtual.

30.3.2 enum serial_manager_status_t

Enumerator

kStatus_SerialManager_Success Success.
kStatus_SerialManager_Error Failed.
kStatus_SerialManager_Busy Busy.
kStatus_SerialManager_Notify Ring buffer is not empty.
kStatus_SerialManager_Canceled the non-blocking request is canceled
kStatus_SerialManager_HandleConflict The handle is opened.
kStatus_SerialManager_RingBufferOverflow The ring buffer is overflowed.

30.4 Function Documentation

30.4.1 serial_manager_status_t SerialManager_Init (serial_handle_t *serialHandle*, serial_manager_config_t ∗ *config*)

This function configures the Serial Manager module with user-defined settings. The user can configure the configuration structure. The parameter serialHandle is a pointer to point to a memory space of size SERIAL_MANAGER_HANDLE_SIZE allocated by the caller. The Serial Manager module supports two types of serial port, UART (includes UART, USART, LPSCI, LPUART, etc) and USB CDC. Please refer to serial_port_type_t for serial port setting. These two types can be set by using serial_manager_config_t.

Example below shows how to use this API to configure the Serial Manager. For UART,

```
*   #define SERIAL_MANAGER_RING_BUFFER_SIZE          (256U)
*   static uint8_t s_serialHandleBuffer[SERIAL_MANAGER_HANDLE_SIZE];
*   static serial_handle_t s_serialHandle = &s_serialHandleBuffer[0];
*   static uint8_t s_ringBuffer[SERIAL_MANAGER_RING_BUFFER_SIZE];
*
*   serial_manager_config_t config;
*   serial_port_uart_config_t uartConfig;
*   config.type = kSerialPort_Uart;
*   config.ringBuffer = &s_ringBuffer[0];
*   config.ringBufferSize = SERIAL_MANAGER_RING_BUFFER_SIZE;
*   uartConfig.instance = 0;
*   uartConfig.clockRate = 24000000;
*   uartConfig.baudRate = 115200;
*   uartConfig.parityMode = kSerialManager_UartParityDisabled;
*   uartConfig.stopBitCount = kSerialManager_UartOneStopBit;
*   uartConfig.enableRx = 1;
*   uartConfig.enableTx = 1;
*   config.portConfig = &uartConfig;
*   SerialManager_Init(s_serialHandle, &config);
*
```

For USB CDC,

```
*   #define SERIAL_MANAGER_RING_BUFFER_SIZE          (256U)
*   static uint8_t s_serialHandleBuffer[SERIAL_MANAGER_HANDLE_SIZE];
*   static serial_handle_t s_serialHandle = &s_serialHandleBuffer[0];
*   static uint8_t s_ringBuffer[SERIAL_MANAGER_RING_BUFFER_SIZE];
*
*   serial_manager_config_t config;
```

```
*   serial_port_usb_cdc_config_t usbCdcConfig;
*   config.type = kSerialPort_UsbCdc;
*   config.ringBuffer = &s_ringBuffer[0];
*   config.ringBufferSize = SERIAL_MANAGER_RING_BUFFER_SIZE;
*   usbCdcConfig.controllerIndex =
      kSerialManager_UsbControllerKhci0;
*   config.portConfig = &usbCdcConfig;
*   SerialManager_Init(s_serialHandle, &config);
*
```

Parameters

serialHandle	Pointer to point to a memory space of size SERIAL_MANAGER_HANDLE_SIZE allocated by the caller.
config	Pointer to user-defined configuration structure.

Return values

kStatus_SerialManager_-Error	An error occurred.
kStatus_SerialManager_-Success	The Serial Manager module initialization succeed.

30.4.2 serial_manager_status_t SerialManager_Deinit (serial_handle_t *serialHandle*)

This function de-initializes the serial manager module instance. If the opened writing or reading handle is not closed, the function will return kStatus_SerialManager_Busy.

Parameters

serialHandle	The serial manager module handle pointer.

Return values

kStatus_SerialManager_-Success	The serial manager de-initialization succeed.
kStatus_SerialManager_-Busy	Opened reading or writing handle is not closed.

30.4.3 serial_manager_status_t SerialManager_OpenWriteHandle (serial_handle_t *serialHandle,* serial_write_handle_t *writeHandle*)

This function Opens a writing handle for the serial manager module. If the serial manager needs to be used in different tasks, the task should open a dedicated write handle for itself by calling SerialManager-_OpenWriteHandle. Since there can only one buffer for transmission for the writing handle at the same time, multiple writing handles need to be opened when the multiple transmission is needed for a task.

Parameters

serialHandle	The serial manager module handle pointer.
writeHandle	The serial manager module writing handle pointer.

Return values

kStatus_SerialManager_-Error	An error occurred.
kStatus_SerialManager_-HandleConflict	The writing handle was opened.
kStatus_SerialManager_-Success	The writing handle is opened.

Example below shows how to use this API to write data. For task 1,

```
*   static uint8_t s_serialWriteHandleBuffer1[SERIAL_MANAGER_WRITE_HANDLE_SIZE
      ];
*   static serial_write_handle_t s_serialWriteHandle1 = &s_serialWriteHandleBuffer1[0];
*   static uint8_t s_nonBlockingWelcome1[] = "This is non-blocking writing log for task1!\r\n";
*   SerialManager_OpenWriteHandle(serialHandle, s_serialWriteHandle1);
*   SerialManager_InstallTxCallback(s_serialWriteHandle1, Task1_SerialManagerTxCallback,
      s_serialWriteHandle1);
*   SerialManager_WriteNonBlocking(s_serialWriteHandle1, s_nonBlockingWelcome1, sizeof(
      s_nonBlockingWelcome1) - 1);
*
```

For task 2,

```
*   static uint8_t s_serialWriteHandleBuffer2[SERIAL_MANAGER_WRITE_HANDLE_SIZE
      ];
*   static serial_write_handle_t s_serialWriteHandle2 = &s_serialWriteHandleBuffer2[0];
*   static uint8_t s_nonBlockingWelcome2[] = "This is non-blocking writing log for task2!\r\n";
*   SerialManager_OpenWriteHandle(serialHandle, s_serialWriteHandle2);
*   SerialManager_InstallTxCallback(s_serialWriteHandle2, Task2_SerialManagerTxCallback,
      s_serialWriteHandle2);
*   SerialManager_WriteNonBlocking(s_serialWriteHandle2, s_nonBlockingWelcome2, sizeof(
      s_nonBlockingWelcome2) - 1);
*
```

30.4.4 serial_manager_status_t SerialManager_CloseWriteHandle (serial_write_handle_t *writeHandle*)

This function Closes a writing handle for the serial manager module.

Parameters

writeHandle	The serial manager module writing handle pointer.

Return values

kStatus_SerialManager_-Success	The writing handle is closed.

30.4.5 serial_manager_status_t SerialManager_OpenReadHandle (serial_handle_t *serialHandle*, serial_read_handle_t *readHandle*)

This function Opens a reading handle for the serial manager module. The reading handle can not be opened multiple at the same time. The error code kStatus_SerialManager_Busy would be returned when the previous reading handle is not closed. And There can only be one buffer for receiving for the reading handle at the same time.

Parameters

serialHandle	The serial manager module handle pointer.
readHandle	The serial manager module reading handle pointer.

Return values

kStatus_SerialManager_-Error	An error occurred.
kStatus_SerialManager_-Success	The reading handle is opened.
kStatus_SerialManager_-Busy	Previous reading handle is not closed.

Example below shows how to use this API to read data.

```
*   static uint8_t s_serialReadHandleBuffer[SERIAL_MANAGER_READ_HANDLE_SIZE];
*   static serial_read_handle_t s_serialReadHandle = &s_serialReadHandleBuffer[0];
*   SerialManager_OpenReadHandle(serialHandle, s_serialReadHandle);
*   static uint8_t s_nonBlockingBuffer[64];
*   SerialManager_InstallRxCallback(s_serialReadHandle, APP_SerialManagerRxCallback, s_serialReadHandle);
*   SerialManager_ReadNonBlocking(s_serialReadHandle, s_nonBlockingBuffer, sizeof(s_nonBlockingBuffer));
*
```

30.4.6 serial_manager_status_t SerialManager_CloseReadHandle (serial_read_handle_t *readHandle*)

This function Closes a reading for the serial manager module.

Parameters

readHandle	The serial manager module reading handle pointer.

Return values

kStatus_SerialManager_-Success	The reading handle is closed.

30.4.7 serial_manager_status_t SerialManager_WriteBlocking (serial-_write_handle_t *writeHandle*, uint8_t * *buffer*, uint32_t *length*)

This is a blocking function, which polls the sending queue, waits for the sending queue to be empty. This function sends data using an interrupt method. The interrupt of the hardware could not be disabled. And There can only one buffer for transmission for the writing handle at the same time.

Note

> The function SerialManager_WriteBlocking and the function #SerialManager_WriteNonBlocking cannot be used at the same time. And, the function #SerialManager_CancelWriting cannot be used to abort the transmission of this function.

Parameters

writeHandle	The serial manager module handle pointer.
buffer	Start address of the data to write.
length	Length of the data to write.

Return values

kStatus_SerialManager_-Success	Successfully sent all data.
kStatus_SerialManager_-Busy	Previous transmission still not finished; data not all sent yet.

kStatus_SerialManager_-Error	An error occurred.

30.4.8 serial_manager_status_t SerialManager_ReadBlocking (serial-_read_handle_t *readHandle*, uint8_t * *buffer*, uint32_t *length*)

This is a blocking function, which polls the receiving buffer, waits for the receiving buffer to be full. This function receives data using an interrupt method. The interrupt of the hardware could not be disabled. And There can only one buffer for receiving for the reading handle at the same time.

Note

> The function SerialManager_ReadBlocking and the function #SerialManager_ReadNonBlocking cannot be used at the same time. And, the function #SerialManager_CancelReading cannot be used to abort the transmission of this function.

Parameters

readHandle	The serial manager module handle pointer.
buffer	Start address of the data to store the received data.
length	The length of the data to be received.

Return values

kStatus_SerialManager_-Success	Successfully received all data.
kStatus_SerialManager_-Busy	Previous transmission still not finished; data not all received yet.
kStatus_SerialManager_-Error	An error occurred.

30.4.9 serial_manager_status_t SerialManager_EnterLowpower (serial_handle_t *serialHandle*)

This function is used to prepare to enter low power consumption.

Parameters

serialHandle	The serial manager module handle pointer.

Return values

kStatus_SerialManager_-Success	Successful operation.

30.4.10 serial_manager_status_t SerialManager_ExitLowpower (serial_handle_t *serialHandle*)

This function is used to restore from low power consumption.

Parameters

serialHandle	The serial manager module handle pointer.

Return values

kStatus_SerialManager_-Success	Successful operation.

30.5 Serial Port Uart

30.5.1 Overview

Data Structures

- struct serial_port_uart_config_t
  serial port uart config struct More...

Macros

- #define SERIAL_PORT_UART_HANDLE_SIZE (4U)
  serial port uart handle size

Enumerations

- enum serial_port_uart_parity_mode_t {
  kSerialManager_UartParityDisabled = 0x0U,
  kSerialManager_UartParityEven = 0x1U,
  kSerialManager_UartParityOdd = 0x2U }
  serial port uart parity mode
- enum serial_port_uart_stop_bit_count_t {
  kSerialManager_UartOneStopBit = 0U,
  kSerialManager_UartTwoStopBit = 1U }
  serial port uart stop bit count

30.5.2 Data Structure Documentation

30.5.2.1 struct serial_port_uart_config_t

Data Fields

- uint32_t clockRate
  clock rate
- uint32_t baudRate
  baud rate
- serial_port_uart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- serial_port_uart_stop_bit_count_t stopBitCount
  Number of stop bits, 1 stop bit (default) or 2 stop bits.
- uint8_t instance
  Instance (0 - UART0, 1 - UART1, ...), detail information please refer to the SOC corresponding RM.
- uint8_t enableRx
  Enable RX.
- uint8_t enableTx

Enable TX.

30.5.2.1.0.32 Field Documentation

30.5.2.1.0.32.1 uint8_t serial_port_uart_config_t::instance

30.5.3 Enumeration Type Documentation

30.5.3.1 enum serial_port_uart_parity_mode_t

Enumerator

kSerialManager_UartParityDisabled Parity disabled.
kSerialManager_UartParityEven Parity even enabled.
kSerialManager_UartParityOdd Parity odd enabled.

30.5.3.2 enum serial_port_uart_stop_bit_count_t

Enumerator

kSerialManager_UartOneStopBit One stop bit.
kSerialManager_UartTwoStopBit Two stop bits.

30.6 Serial Port USB

30.6.1 Overview

Modules

- USB Device Configuration

Data Structures

- struct serial_port_usb_cdc_config_t
  serial port usb config struct More...

Macros

- #define SERIAL_PORT_USB_CDC_HANDLE_SIZE (72)
  serial port usb handle size
- #define USB_DEVICE_INTERRUPT_PRIORITY (3U)
  USB interrupt priority.

Enumerations

- enum serial_port_usb_cdc_controller_index_t {
  kSerialManager_UsbControllerKhci0 = 0U,
  kSerialManager_UsbControllerKhci1 = 1U,
  kSerialManager_UsbControllerEhci0 = 2U,
  kSerialManager_UsbControllerEhci1 = 3U,
  kSerialManager_UsbControllerLpcIp3511Fs0 = 4U,
  kSerialManager_UsbControllerLpcIp3511Fs1 = 5U,
  kSerialManager_UsbControllerLpcIp3511Hs0 = 6U,
  kSerialManager_UsbControllerLpcIp3511Hs1 = 7U,
  kSerialManager_UsbControllerOhci0 = 8U,
  kSerialManager_UsbControllerOhci1 = 9U,
  kSerialManager_UsbControllerIp3516Hs0 = 10U,
  kSerialManager_UsbControllerIp3516Hs1 = 11U }
  USB controller ID.

30.6.2 Data Structure Documentation

30.6.2.1 struct serial_port_usb_cdc_config_t

Data Fields

- serial_port_usb_cdc_controller_index_t controllerIndex
  controller index

30.6.3 Enumeration Type Documentation

30.6.3.1 enum serial_port_usb_cdc_controller_index_t

Enumerator

kSerialManager_UsbControllerKhci0 KHCI 0U.
kSerialManager_UsbControllerKhci1 KHCI 1U, Currently, there are no platforms which have two KHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbControllerEhci0 EHCI 0U.
kSerialManager_UsbControllerEhci1 EHCI 1U, Currently, there are no platforms which have two EHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbControllerLpcIp3511Fs0 LPC USB IP3511 FS controller 0.
kSerialManager_UsbControllerLpcIp3511Fs1 LPC USB IP3511 FS controller 1, there are no platforms which have two IP3511 IPs, this is reserved to be used in the future.
kSerialManager_UsbControllerLpcIp3511Hs0 LPC USB IP3511 HS controller 0.
kSerialManager_UsbControllerLpcIp3511Hs1 LPC USB IP3511 HS controller 1, there are no platforms which have two IP3511 IPs, this is reserved to be used in the future.
kSerialManager_UsbControllerOhci0 OHCI 0U.
kSerialManager_UsbControllerOhci1 OHCI 1U, Currently, there are no platforms which have two OHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbControllerIp3516Hs0 IP3516HS 0U.
kSerialManager_UsbControllerIp3516Hs1 IP3516HS 1U, Currently, there are no platforms which have two IP3516HS IPs, this is reserved to be used in the future.

30.6.4 USB Device Configuration

30.6.4.1 Overview

Macros

- #define USB_DEVICE_CONFIG_SELF_POWER (1U)
  Whether device is self power.
- #define USB_DEVICE_CONFIG_ENDPOINTS (4U)
  How many endpoints are supported in the stack.
- #define USB_DEVICE_CONFIG_USE_TASK (0U)
  Whether the device task is enabled.
- #define USB_DEVICE_CONFIG_MAX_MESSAGES (8U)
  How many the notification message are supported when the device task is enabled.
- #define USB_DEVICE_CONFIG_USB20_TEST_MODE (0U)
  Whether test mode enabled.
- #define USB_DEVICE_CONFIG_CV_TEST (0U)
  Whether device CV test is enabled.
- #define USB_DEVICE_CONFIG_COMPLIANCE_TEST (0U)
  Whether device compliance test is enabled.
- #define USB_DEVICE_CONFIG_KEEP_ALIVE_MODE (0U)
  Whether the keep alive feature enabled.
- #define USB_DEVICE_CONFIG_BUFFER_PROPERTY_CACHEABLE (0U)
  Whether the transfer buffer is cache-enabled or not.
- #define USB_DEVICE_CONFIG_LOW_POWER_MODE (0U)
  Whether the low power mode is enabled or not.
- #define USB_DEVICE_CONFIG_REMOTE_WAKEUP (0U)
  The device remote wakeup is unsupported.
- #define USB_DEVICE_CONFIG_DETACH_ENABLE (0U)
  Whether the device detached feature is enabled or not.
- #define USB_DEVICE_CONFIG_ERROR_HANDLING (0U)
  Whether handle the USB bus error.
- #define USB_DEVICE_CHARGER_DETECT_ENABLE (0U)
  Whether the device charger detect feature is enabled or not.

class instance define

- #define USB_DEVICE_CONFIG_HID (0U)
  HID instance count.
- #define USB_DEVICE_CONFIG_CDC_ACM (1U)
  CDC ACM instance count.
- #define USB_DEVICE_CONFIG_MSC (0U)
  MSC instance count.
- #define USB_DEVICE_CONFIG_AUDIO (0U)
  Audio instance count.
- #define USB_DEVICE_CONFIG_PHDC (0U)
  PHDC instance count.
- #define USB_DEVICE_CONFIG_VIDEO (0U)
  Video instance count.
- #define USB_DEVICE_CONFIG_CCID (0U)

CCID instance count.
- #define USB_DEVICE_CONFIG_PRINTER (0U)
  Printer instance count.
- #define USB_DEVICE_CONFIG_DFU (0U)
  DFU instance count.

30.6.4.2 Macro Definition Documentation

30.6.4.2.1 #define USB_DEVICE_CONFIG_SELF_POWER (1U)

1U supported, 0U not supported

30.6.4.2.2 #define USB_DEVICE_CONFIG_ENDPOINTS (4U)

30.6.4.2.3 #define USB_DEVICE_CONFIG_USE_TASK (0U)

30.6.4.2.4 #define USB_DEVICE_CONFIG_MAX_MESSAGES (8U)

30.6.4.2.5 #define USB_DEVICE_CONFIG_USB20_TEST_MODE (0U)

30.6.4.2.6 #define USB_DEVICE_CONFIG_CV_TEST (0U)

30.6.4.2.7 #define USB_DEVICE_CONFIG_COMPLIANCE_TEST (0U)

If the macro is enabled, the test mode and CV test macroes will be set.

30.6.4.2.8 #define USB_DEVICE_CONFIG_KEEP_ALIVE_MODE (0U)

30.6.4.2.9 #define USB_DEVICE_CONFIG_BUFFER_PROPERTY_CACHEABLE (0U)

30.6.4.2.10 #define USB_DEVICE_CONFIG_LOW_POWER_MODE (0U)

30.6.4.2.11 #define USB_DEVICE_CONFIG_REMOTE_WAKEUP (0U)

30.6.4.2.12 #define USB_DEVICE_CONFIG_DETACH_ENABLE (0U)

30.6.4.2.13 #define USB_DEVICE_CONFIG_ERROR_HANDLING (0U)

30.6.4.2.14 #define USB_DEVICE_CHARGER_DETECT_ENABLE (0U)

30.7 Serial Port SWO

30.7.1 Overview

Data Structures

- struct serial_port_swo_config_t
  serial port swo config struct More...

Macros

- #define SERIAL_PORT_SWO_HANDLE_SIZE (12U)
  serial port swo handle size

Enumerations

- enum serial_port_swo_protocol_t {
  kSerialManager_SwoProtocolManchester = 1U,
  kSerialManager_SwoProtocolNrz = 2U }
  serial port swo protocol

30.7.2 Data Structure Documentation

30.7.2.1 struct serial_port_swo_config_t

Data Fields

- uint32_t clockRate
  clock rate
- uint32_t baudRate
  baud rate
- uint32_t port
  Port used to transfer data.
- serial_port_swo_protocol_t protocol
  SWO protocol.

30.7.3 Enumeration Type Documentation

30.7.3.1 enum serial_port_swo_protocol_t

Enumerator

kSerialManager_SwoProtocolManchester SWO Manchester protocol.
kSerialManager_SwoProtocolNrz SWO UART/NRZ protocol.

30.8 Serial Port Virtual USB

30.8.1 Overview

This chapter describes how to redirect the serial manager stream to application CDC. The weak functions can be implemented by application to redirect the serial manager stream. The weak functions are following,

USB_DeviceVcomInit - Initialize the cdc vcom.

USB_DeviceVcomDeinit - De-initialize the cdc vcom.

USB_DeviceVcomWrite - Write data with non-blocking mode. After data is sent, the installed TX callback should be called with the result.

USB_DeviceVcomRead - Read data with non-blocking mode. After data is received, the installed RX callback should be called with the result.

USB_DeviceVcomCancelWrite - Cancel write request.

USB_DeviceVcomInstallTxCallback - Install TX callback.

USB_DeviceVcomInstallRxCallback - Install RX callback.

USB_DeviceVcomIsrFunction - The hardware ISR function.

Data Structures

- struct serial_port_usb_cdc_virtual_config_t
  serial port usb config struct More...

Macros

- #define SERIAL_PORT_USB_VIRTUAL_HANDLE_SIZE (40U)
  serial port USB handle size

Enumerations

- enum serial_port_usb_cdc_virtual_controller_index_t {
  kSerialManager_UsbVirtualControllerKhci0 = 0U,
  kSerialManager_UsbVirtualControllerKhci1 = 1U,
  kSerialManager_UsbVirtualControllerEhci0 = 2U,
  kSerialManager_UsbVirtualControllerEhci1 = 3U,
  kSerialManager_UsbVirtualControllerLpcIp3511Fs0 = 4U,
  kSerialManager_UsbVirtualControllerLpcIp3511Fs1,
  kSerialManager_UsbVirtualControllerLpcIp3511Hs0 = 6U,
  kSerialManager_UsbVirtualControllerLpcIp3511Hs1,
  kSerialManager_UsbVirtualControllerOhci0 = 8U,
  kSerialManager_UsbVirtualControllerOhci1 = 9U,
  kSerialManager_UsbVirtualControllerIp3516Hs0 = 10U,
  kSerialManager_UsbVirtualControllerIp3516Hs1 = 11U }
  USB controller ID.

Variables

- serial_port_usb_cdc_virtual_controller_index_t serial_port_usb_cdc_virtual_config_t::controllerIndex
  controller index

30.8.2 Data Structure Documentation

30.8.2.1 struct serial_port_usb_cdc_virtual_config_t

Data Fields

- serial_port_usb_cdc_virtual_controller_index_t controllerIndex
  controller index

30.8.3 Enumeration Type Documentation

30.8.3.1 enum serial_port_usb_cdc_virtual_controller_index_t

Enumerator

kSerialManager_UsbVirtualControllerKhci0 KHCI 0U.
kSerialManager_UsbVirtualControllerKhci1 KHCI 1U, Currently, there are no platforms which have two KHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbVirtualControllerEhci0 EHCI 0U.

kSerialManager_UsbVirtualControllerEhci1 EHCI 1U, Currently, there are no platforms which have two EHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbVirtualControllerLpcIp3511Fs0 LPC USB IP3511 FS controller 0.
kSerialManager_UsbVirtualControllerLpcIp3511Fs1 LPC USB IP3511 FS controller 1, there are no platforms which have two IP3511 IPs, this is reserved to be used in the future.
kSerialManager_UsbVirtualControllerLpcIp3511Hs0 LPC USB IP3511 HS controller 0.
kSerialManager_UsbVirtualControllerLpcIp3511Hs1 LPC USB IP3511 HS controller 1, there are no platforms which have two IP3511 IPs, this is reserved to be used in the future.
kSerialManager_UsbVirtualControllerOhci0 OHCI 0U.
kSerialManager_UsbVirtualControllerOhci1 OHCI 1U, Currently, there are no platforms which have two OHCI IPs, this is reserved to be used in the future.
kSerialManager_UsbVirtualControllerIp3516Hs0 IP3516HS 0U.
kSerialManager_UsbVirtualControllerIp3516Hs1 IP3516HS 1U, Currently, there are no platforms which have two IP3516HS IPs, this is reserved to be used in the future.

Chapter 31
GenericList

31.1 Overview

Data Structures

- struct list_handle_t
  The list structure. More...
- struct list_element_handle_t
  The list element. More...

Enumerations

- enum list_status_t {
  kLIST_Ok = kStatus_Success,
  kLIST_DuplicateError = MAKE_STATUS(kStatusGroup_LIST, 1),
  kLIST_Full = MAKE_STATUS(kStatusGroup_LIST, 2),
  kLIST_Empty = MAKE_STATUS(kStatusGroup_LIST, 3),
  kLIST_OrphanElement = MAKE_STATUS(kStatusGroup_LIST, 4) }

Functions

- void LIST_Init (list_handle_t list, uint32_t max)

- list_handle_t LIST_GetList (list_element_handle_t element)
  Gets the list that contains the given element.
- list_status_t LIST_AddHead (list_handle_t list, list_element_handle_t element)
  Links element to the head of the list.
- list_status_t LIST_AddTail (list_handle_t list, list_element_handle_t element)
  Links element to the tail of the list.
- list_element_handle_t LIST_RemoveHead (list_handle_t list)
  Unlinks element from the head of the list.
- list_element_handle_t LIST_GetHead (list_handle_t list)
  Gets head element handle.
- list_element_handle_t LIST_GetNext (list_element_handle_t element)
  Gets next element handle for given element handle.
- list_element_handle_t LIST_GetPrev (list_element_handle_t element)
  Gets previous element handle for given element handle.
- list_status_t LIST_RemoveElement (list_element_handle_t element)
  Unlinks an element from its list.
- list_status_t LIST_AddPrevElement (list_element_handle_t element, list_element_handle_t newElement)
  Links an element in the previous position relative to a given member of a list.
- uint32_t LIST_GetSize (list_handle_t list)
  Gets the current size of a list.

- uint32_t LIST_GetAvailableSize (list_handle_t list)
  Gets the number of free places in the list.

31.2 Data Structure Documentation

31.2.1 struct list_t

Data Fields

- struct list_element_tag ∗ head
  list head
- struct list_element_tag ∗ tail
  list tail
- uint16_t size
  list size
- uint16_t max
  list max number of elements

31.2.2 struct list_element_t

Data Fields

- struct list_element_tag ∗ next
  next list element
- struct list_element_tag ∗ prev
  previous list element
- struct list_tag ∗ list
  pointer to the list

31.3 Enumeration Type Documentation

31.3.1 enum list_status_t

Include

Public macro definitions

Public type definitions

The list status

Enumerator

kLIST_Ok Success.
kLIST_DuplicateError Duplicate Error.
kLIST_Full FULL.
kLIST_Empty Empty.
kLIST_OrphanElement Orphan Element.

31.4 Function Documentation

31.4.1 void LIST_Init (list_handle_t *list*, uint32_t *max*)

Public prototypes

Initialize the list.

This function initialize the list.

Parameters

list	- List handle to initialize.
max	- Maximum number of elements in list. 0 for unlimited.

31.4.2 list_handle_t LIST_GetList (list_element_handle_t *element*)

Parameters

element	- Handle of the element.

Return values

NULL	if element is orphan, Handle of the list the element is inserted into.

31.4.3 list_status_t LIST_AddHead (list_handle_t *list*, list_element_handle_t *element*)

Parameters

list	- Handle of the list.
element	- Handle of the element.

Return values

kLIST_Full	if list is full, kLIST_Ok if insertion was successful.

31.4.4 list_status_t LIST_AddTail (list_handle_t *list*, list_element_handle_t *element*)

Parameters

list	- Handle of the list.
element	- Handle of the element.

Return values

kLIST_Full	if list is full, kLIST_Ok if insertion was successful.

31.4.5 list_element_handle_t LIST_RemoveHead (list_handle_t *list*)

Parameters

list	- Handle of the list.

Return values

NULL	if list is empty, handle of removed element(pointer) if removal was successful.

31.4.6 list_element_handle_t LIST_GetHead (list_handle_t *list*)

Parameters

list	- Handle of the list.

Return values

NULL	if list is empty, handle of removed element(pointer) if removal was successful.

31.4.7 list_element_handle_t LIST_GetNext (list_element_handle_t *element*)

Parameters

element	- Handle of the element.

Return values

NULL	if list is empty, handle of removed element(pointer) if removal was successful.

31.4.8 list_element_handle_t LIST_GetPrev (list_element_handle_t *element*)

Parameters

element	- Handle of the element.

Return values

NULL	if list is empty, handle of removed element(pointer) if removal was successful.

31.4.9 list_status_t LIST_RemoveElement (list_element_handle_t *element*)

Parameters

element	- Handle of the element.

Return values

kLIST_OrphanElement	if element is not part of any list.
kLIST_Ok	if removal was successful.

31.4.10 list_status_t LIST_AddPrevElement (list_element_handle_t *element,* list_element_handle_t *newElement*)

Parameters

element	- Handle of the element.
newElement	- New element to insert before the given member.

Return values

kLIST_OrphanElement	if element is not part of any list.
kLIST_Ok	if removal was successful.

31.4.11 uint32_t LIST_GetSize (list_handle_t *list*)

Parameters

list	- Handle of the list.

Return values

Current	size of the list.

31.4.12 uint32_t LIST_GetAvailableSize (list_handle_t *list*)

Parameters

list	- Handle of the list.

Return values

Available	size of the list.

Chapter 32
UART_Adapter

32.1 Overview

Data Structures

- struct hal_uart_config_t
  UART configuration structure. More...
- struct hal_uart_transfer_t
  UART transfer structure. More...

Macros

- #define UART_ADAPTER_NON_BLOCKING_MODE (0U)
  Enable or disable UART adapter non-blocking mode (1 - enable, 0 - disable)
- #define HAL_UART_TRANSFER_MODE (0U)
  Whether enable transactional function of the UART.

Typedefs

- typedef void(∗ hal_uart_transfer_callback_t)(hal_uart_handle_t handle, hal_uart_status_t status, void ∗callbackParam)
  UART transfer callback function.

Enumerations

- enum hal_uart_status_t {
  kStatus_HAL_UartSuccess = kStatus_Success,
  kStatus_HAL_UartTxBusy = MAKE_STATUS(kStatusGroup_HAL_UART, 1),
  kStatus_HAL_UartRxBusy = MAKE_STATUS(kStatusGroup_HAL_UART, 2),
  kStatus_HAL_UartTxIdle = MAKE_STATUS(kStatusGroup_HAL_UART, 3),
  kStatus_HAL_UartRxIdle = MAKE_STATUS(kStatusGroup_HAL_UART, 4),
  kStatus_HAL_UartBaudrateNotSupport,
  kStatus_HAL_UartProtocolError,
  kStatus_HAL_UartError = MAKE_STATUS(kStatusGroup_HAL_UART, 7) }
  UART status.
- enum hal_uart_parity_mode_t {
  kHAL_UartParityDisabled = 0x0U,
  kHAL_UartParityEven = 0x1U,
  kHAL_UartParityOdd = 0x2U }
  UART parity mode.
- enum hal_uart_stop_bit_count_t {
  kHAL_UartOneStopBit = 0U,
  kHAL_UartTwoStopBit = 1U }
  UART stop bit count.

Initialization and deinitialization

- hal_uart_status_t HAL_UartInit (hal_uart_handle_t handle, hal_uart_config_t *config)
  Initializes a UART instance with the UART handle and the user configuration structure.
- hal_uart_status_t HAL_UartDeinit (hal_uart_handle_t handle)
  Deinitializes a UART instance.

Blocking bus Operations

- hal_uart_status_t HAL_UartReceiveBlocking (hal_uart_handle_t handle, uint8_t *data, size_t length)
  Reads RX data register using a blocking method.
- hal_uart_status_t HAL_UartSendBlocking (hal_uart_handle_t handle, const uint8_t *data, size_t length)
  Writes to the TX register using a blocking method.

32.2 Data Structure Documentation

32.2.1 struct hal_uart_config_t

Data Fields

- uint32_t srcClock_Hz
  Source clock.
- uint32_t baudRate_Bps
  Baud rate.
- hal_uart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- hal_uart_stop_bit_count_t stopBitCount
  Number of stop bits, 1 stop bit (default) or 2 stop bits.
- uint8_t enableRx
  Enable RX.
- uint8_t enableTx
  Enable TX.
- uint8_t instance
  Instance (0 - UART0, 1 - UART1, ...), detail information please refer to the SOC corresponding RM.

32.2.1.0.0.1 Field Documentation

32.2.1.0.0.1.1 uint8_t hal_uart_config_t::instance

Invalid instance value will cause initialization failure.

32.2.2 struct hal_uart_transfer_t

Data Fields

- uint8_t * data

The buffer of data to be transfer.

- size_t dataSize
  The byte count to be transfer.

32.2.2.0.0.2 Field Documentation

32.2.2.0.0.2.1 uint8_t* hal_uart_transfer_t::data

32.2.2.0.0.2.2 size_t hal_uart_transfer_t::dataSize

32.3 Macro Definition Documentation

32.3.1 #define HAL_UART_TRANSFER_MODE (0U)

(0 - disable, 1 - enable)

32.4 Typedef Documentation

32.4.1 typedef void(* hal_uart_transfer_callback_t)(hal_uart_handle_t handle, hal_uart_status_t status, void *callbackParam)

32.5 Enumeration Type Documentation

32.5.1 enum hal_uart_status_t

Enumerator

kStatus_HAL_UartSuccess Successfully.
kStatus_HAL_UartTxBusy TX busy.
kStatus_HAL_UartRxBusy RX busy.
kStatus_HAL_UartTxIdle HAL UART transmitter is idle.
kStatus_HAL_UartRxIdle HAL UART receiver is idle.
kStatus_HAL_UartBaudrateNotSupport Baudrate is not support in current clock source.
kStatus_HAL_UartProtocolError Error occurs for Noise, Framing, Parity, etc. For transactional transfer, The up layer needs to abort the transfer and then starts again
kStatus_HAL_UartError Error occurs on HAL UART.

32.5.2 enum hal_uart_parity_mode_t

Enumerator

kHAL_UartParityDisabled Parity disabled.
kHAL_UartParityEven Parity even enabled.
kHAL_UartParityOdd Parity odd enabled.

32.5.3 enum hal_uart_stop_bit_count_t

Enumerator

kHAL_UartOneStopBit One stop bit.
kHAL_UartTwoStopBit Two stop bits.

32.6 Function Documentation

32.6.1 hal_uart_status_t HAL_UartInit (hal_uart_handle_t *handle*, hal_uart_config_t ∗ *config*)

This function configures the UART module with user-defined settings. The user can configure the configuration structure. The parameter handle is a pointer to point to a memory space of size #HAL_UART_HANDLE_SIZE allocated by the caller. Example below shows how to use this API to configure the UART.

```
*   uint8_t g_UartHandleBuffer[HAL_UART_HANDLE_SIZE];
*   hal_uart_handle_t g_UartHandle = &g_UartHandleBuffer[0];
*   hal_uart_config_t config;
*   config.srcClock_Hz = 48000000;
*   config.baudRate_Bps = 115200U;
*   config.parityMode = kHAL_UartParityDisabled;
*   config.stopBitCount = kHAL_UartOneStopBit;
*   config.enableRx = 1;
*   config.enableTx = 1;
*   config.instance = 0;
*   HAL_UartInit(g_UartHandle, &config);
*
```

Parameters

handle	Pointer to point to a memory space of size #HAL_UART_HANDLE_SIZE allocated by the caller.
config	Pointer to user-defined configuration structure.

Return values

kStatus_HAL_Uart-BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_HAL_Uart-Success	UART initialization succeed

32.6.2 hal_uart_status_t HAL_UartDeinit (hal_uart_handle_t *handle*)

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

handle	UART handle pointer.

Return values

kStatus_HAL_UartSuccess	UART de-initialization succeed

32.6.3 hal_uart_status_t HAL_UartReceiveBlocking (hal_uart_handle_t *handle*, 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 RX register.

Note

> The function HAL_UartReceiveBlocking and the function #HAL_UartTransferReceiveNonBlocking cannot be used at the same time. And, the function #HAL_UartTransferAbortReceive cannot be used to abort the transmission of this function.

Parameters

handle	UART handle pointer.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_HAL_UartError	An error occurred while receiving data.
kStatus_HAL_UartParityError	A parity error occurred while receiving data.
kStatus_HAL_UartSuccess	Successfully received all data.

32.6.4 hal_uart_status_t HAL_UartSendBlocking (hal_uart_handle_t *handle*, 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.

Note

The function HAL_UartSendBlocking and the function #HAL_UartTransferSendNonBlocking cannot be used at the same time. And, the function #HAL_UartTransferAbortSend cannot be used to abort the transmission of this function.

Parameters

handle	UART handle pointer.
data	Start address of the data to write.
length	Size of the data to write.

Return values

kStatus_HAL_Uart-Success	Successfully sent all data.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

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 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.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

arm